spead2

Release 4.3.1

National Research Foundation (SARAO)

May 29, 2024

CONTENTS

1 Introduction to spead2 **3**
- 1.1 Preparation . . . 3
- 1.2 Installing spead2 for Python . . . 4
- 1.3 Installing spead2 for C++ . . . 4

2 Python API for spead2 **7**
- 2.1 SPEAD flavours . . . 7
- 2.2 Mapping of SPEAD protocol to Python . . . 8
- 2.3 Stream control items . . . 8
- 2.4 Items and item groups . . . 9
- 2.5 Thread pools . . . 11
- 2.6 Receiving . . . 11
- 2.7 Sending . . . 19
- 2.8 In-process transport . . . 27
- 2.9 Logging . . . 28
- 2.10 Support for ibverbs . . . 28
- 2.11 Chunking receiver . . . 31
- 2.12 Chunking stream groups . . . 35

3 C++ API for spead2 **37**
- 3.1 C++ API stability . . . 37
- 3.2 Asynchronous I/O . . . 37
- 3.3 Receiving . . . 39
- 3.4 Sending . . . 57
- 3.5 In-process transport . . . 67
- 3.6 Logging . . . 68
- 3.7 Support for ibverbs . . . 68
- 3.8 Chunking receiver . . . 73
- 3.9 Chunking stream groups . . . 80

4 Advanced features **87**
- 4.1 Chunking receiver . . . 87
- 4.2 Chunking stream groups . . . 89
- 4.3 Receiver stream statistics . . . 90

5 Performance tuning **93**
- 5.1 System tuning . . . 93
- 5.2 Protocol design . . . 95
- 5.3 Application tuning . . . 96

6 Command-line tools **99**

6.1 spead2_bench . . . 99
6.2 spead2_send/spead2_recv . . . 99
6.3 mcdump . . . 99
6.4 spead2_net_raw . . . 101

7 Migrating to version 3 **103**
7.1 Receive stream configuration . . . 103
7.2 Send stream configuration . . . 104
7.3 Substreams . . . 104
7.4 Out-of-order packets . . . 104
7.5 Loop argument to asyncio functions . . . 104
7.6 Command-line arguments in tools . . . 105
7.7 Removal of deprecated functionality . . . 105
7.8 Queue depth for sending with ibverbs . . . 105

8 Migrating to version 4 **107**
8.1 Removed functionality . . . 107
8.2 Meson . . . 108
8.3 C++17 . . . 109
8.4 Boost . . . 110
8.5 pcap . . . 110
8.6 Code generation . . . 110
8.7 Python configuration . . . 110
8.8 Python editable installs . . . 110

9 Developer documentation **111**
9.1 Development processes . . . 111
9.2 Design . . . 116

10 Changelog **125**

11 License **149**

12 Indices and tables **151**

Python Module Index **153**

Index **155**

Contents:

CHAPTER

ONE

INTRODUCTION TO SPEAD2

spead2 is an implementation of the SPEAD protocol, with both Python and C++ bindings. The *2* in the name indicates that this is a new implementation of the protocol; the protocol remains essentially the same. Compared to the PySPEAD implementation, spead2:

- is at least an order of magnitude faster when dealing with large heaps;
- correctly implements several aspects of the protocol that were implemented incorrectly in PySPEAD (bug-compatibility is also available);
- correctly implements many corner cases on which PySPEAD would simply fail;
- cleanly supports several SPEAD flavours (e.g. 64-40 and 64-48) in one module, with the receiver adapting to the flavour used by the sender;
- supports Python 3;
- supports asynchronous operation, using asyncio.

1.1 Preparation

There is optional support for ibverbs for higher performance, and pcap for reading from previously captured packet dumps. If the libraries (including development headers) are installed, they will be detected automatically and support for them will be included.

High-performance usage requires larger buffer sizes than Linux allows by default. The following commands will increase the permitted buffer sizes on Linux:

```
sysctl net.core.wmem_max=16777216
sysctl net.core.rmem_max=16777216
```

Note that these commands are not persistent across reboots, and the settings need to be stored in `/etc/sysctl.conf` or `/etc/sysctl.d`.

1.2 Installing spead2 for Python

The only Python dependency is numpy.

The test suite has additional dependencies; refer to *Getting started with development* if you are developing spead2.

There are two ways to install spead2 for Python: compiling from source and installing a binary wheel.

1.2.1 Installing a binary wheel

As from version 1.12, binary wheels are provided on PyPI for x86-64 Linux systems. These support all the optional features, and it is the recommended installation method as it does not depend on a compiler, development libraries etc. The wheels use the "manylinux2014" tag, which requires at least **pip** 19.3 to install.

Since version 4.0 there are also aarch64 Linux wheels, which use the "manylinux_2_28" tag and require at least **pip** 20.3; and wheels for MacOS (both Intel and Apple Silicon).

Provided your system meets these requirements, just run:

```
pip install spead2
```

1.2.2 Python install from source

Installing from source requires a modern C++ compiler supporting C++17 (GCC 7+ or Clang 4+, although GCC 9.4 and Clang 10 are the oldest tested versions and support for older compilers may be dropped) as well as Boost 1.69+ (only headers are required), libdivide, and the Python development headers. At the moment only GNU/Linux and OS X get tested but other POSIX-like systems should work too. There are no plans to support Windows.

Installation works with standard Python installation methods.

1.3 Installing spead2 for C++

Installing spead2 requires

- a modern C++ compiler supporting C++17 (see above for supported compilers)
- Boost 1.69+, including the compiled boost_program_options library
- libdivide
- Python 3.x, with the packaging, jinja2, and pycparser packages
- Meson 1.2 or later (note that this might be newer than the Meson package in your operating system's package manager).

At the moment only GNU/Linux and OS X get tested but other POSIX-like systems should work too. There are no plans to support Windows.

Compilation uses the standard Meson flow (refer to the Meson manual for further help):

```
meson setup [options] build
cd build
meson compile
meson install
```

Optional features are autodetected by default, but can be disabled using Meson options. To see the available options, run **meson configure** in the build directory. One option that may squeeze out a very small amount of extra performance is link-time optimization, enabled with `-Db_lto=true`.

The installation will install some benchmark tools, a static library, and the header files.

1.3.1 Shared library

There is experimental support for building a shared library. Pass `--default_library=both` to `meson setup`. It's also possible to pass `--default_library=shared`, in which case the static library will not be built, and the command-line tools will be linked against the shared library.

It's not recommended for general use because the binary interface is likely to be incompatible between spead2 versions, requiring software linked against the shared library to be recompiled after upgrading spead2 (which defeats one of the points of a shared library). It also exports a lot of symbols (e.g., from Boost) that may clash with other libraries. Performance may be lower than using the static library. It is made available for users who need to load the library dynamically as part of a plugin system.

CHAPTER

TWO

PYTHON API FOR SPEAD2

This documentation does not cover all the classes and methods in the module. Instead, it documents those that are expected to be commonly used by the user, and omits those designed for the classes to communicate with each other or with the C++ backend.

2.1 SPEAD flavours

The SPEAD protocol is versioned and within a version allows for multiple *flavours*, with different numbers of bits for item pointer fields. The spead2 library supports all SPEAD-64-*XX* flavours of version 4, where *XX* is a multiple of 8.

Furthermore, PySPEAD 0.5.2 has a number of bugs in its implementation of the protocol, which effectively defines a new protocol. This is treated as part of the flavour in spead2. Some receive functions have a *bug_compat* parameter which specifies which of these bugs to maintain compatibility with:

- `spead2.BUG_COMPAT_DESCRIPTOR_WIDTHS`: the descriptors are encoded with shape and format fields sized as for SPEAD-64-40, regardless of the actual flavour.
- `spead2.BUG_COMPAT_SHAPE_BIT_1`: the first byte of a shape is set to 2 to indicate a variably-sized dimension, instead of 1.
- `spead2.BUG_COMPAT_SWAP_ENDIAN`: numpy arrays are encoded/decoded in the opposite endianness to that specified in the descriptor.
- `spead2.BUG_COMPAT_NO_SCALAR_NUMPY`: scalar items specified with a descriptor are transmitted with a descriptor, even if it is possible to convert it to a dtype.
- `spead2.BUG_COMPAT_PYSPEAD_0_5_2`: all of the above (and any other bugs later found in this version of PySPEAD).

For sending, the full flavour is specified by a `spead2.Flavour` object. It allows all the fields to be specified to allow for future expansion, but `ValueError` is raised unless *version* is 4 and *item_pointer_bits* is 64. There is a default constructor that returns SPEAD-64-40 with bug compatibility disabled.

class Flavour(*version*, *item_pointer_bits*, *heap_address_bits*, *bug_compat=0*)

The constructor arguments are available as read-only attributes.

2.2 Mapping of SPEAD protocol to Python

- Any descriptor with a numpy header is handled by numpy. The value is converted to native endian, but is otherwise left untouched.
- Strings are expected to use ASCII encoding only. At present this is variably enforced. Future versions may apply stricter enforcement. This applies to names, descriptions, and to values passed with the *c* format code.
- The *c* format code may only be used with length 8, and *f* may only be used with lengths 32 or 64.
- The *0* format code is not supported.
- All values sent or received are converted to numpy arrays. If the descriptor uses a numpy header, this is the type of the array. Otherwise, a dtype is constructed by converting the format code. The following are converted to numpy primitive types:
  - u8, u16, u32, u64
  - i8, i16, i32, i64
  - f32, f64
  - b8 (converted to dtype bool)
  - c8 (converted to dtype S1)

  Other fields will be kept as Python objects. If there are multiple fields, their names will be generated by numpy (*f0*, *f1*, etc). If all the fields convert to native types, a fast path will be used for sending and receiving (as fast as using an explicit numpy header).
- At most one element of the shape may indicate a variable-length field, whose length will be computed from the size of the item, or zero if any other element of the shape is zero.

When transmitting data, a few cases are handled specially:

- If the expected shape is one-dimensional, but the provided value is an instance of `bytes`, `str` or `unicode`, it will be broken up into its individual characters. This is a convenience for sending variable-length strings.
- If the format is a single signed or unsigned integer whose number of bits is less than 64 but a multiple of 8, and the value is a zero-dimensional numpy array with dtype `>u8`, the relevant bytes are referenced by the heap. The value can later be updated and the same heap sent again without creating a new `Heap` object.

When receiving data, some transformations are made:

- A zero-dimensional array is returned as a scalar, rather than a zero-dimensional array object.
- If the format is given and is c8 and the array is one-dimensional, it is joined together into a Python `str`.

2.3 Stream control items

A heap with the `CTRL_STREAM_STOP` flag will shut down the stream, but the heap is not passed on to the application. Senders should thus avoid putting any other data in such heaps. These heaps are not automatically sent; use `spead2.send.HeapGenerator.get_end()` to produce such a heap.

In contrast, stream start flags (`CTRL_STREAM_START`) have no effect on internal processing. Senders can generate them using `spead2.send.HeapGenerator.get_start()` and receivers can detect them using `spead2.recv.Heap.is_start_of_stream()`.

2.4 Items and item groups

Each data item that can be communicated over SPEAD is described by a `spead2.Descriptor`. Items combine a descriptor with a current value, and a version number that is used to detect which items have been changed (either in the library when transmitting, or by the user when receiving).

class spead2.**Descriptor**(*id*, *name*, *description*, *shape*, *dtype=None*, *order='C'*, *format=None*)

Metadata for a SPEAD item.

There are a number of restrictions on the parameters, which will cause *ValueError* to be raised if violated:

- At most one element of *shape* can be *None*.
- Exactly one of *dtype* and *format* must be non-*None*.
- If *dtype* is specified, *shape* cannot have any unknown dimensions.
- If *format* is specified, *order* must be 'C'
- If *dtype* is specified, it cannot contain objects, and must have positive size.

Parameters

- **id** (*int*) – SPEAD item ID
- **name** (*str*) – Short item name, suitable for use as a key
- **description** (*str*) – Long item description
- **shape** (*sequence*) – Dimensions, with *None* indicating a variable-size dimension
- **dtype** (*numpy data type, optional*) – Data type, or *None* if *format* will be used instead
- **order** (*{'C', 'F'}*) – Indicates C-order or Fortran-order storage
- **format** (*list of pairs, optional*) – Structure fields for generic (non-numpy) type. Each element of the list is a tuple of field code and bit length.

itemsize_bits

Number of bits per element

is_variable_size()

Determine whether any element of the size is dynamic

dynamic_shape(*max_elements*)

Determine the dynamic shape, given incoming data that is big enough to hold *max_elements* elements.

compatible_shape(*shape*)

Determine whether *shape* is compatible with the (possibly variable-sized) shape for this descriptor

class spead2.**Item**(**args*, ***kwargs*, *value=None*)

A SPEAD item with a value and a version number.

Parameters

value (*object, optional*) – Initial value

value

Current value. Assigning to this will increment the version number. Assigning `None` will raise `ValueError` because there is no way to encode this using SPEAD.

Warning: If you modify a mutable value in-place, the change will not be detected, and by default *HeapGenerator.get_heap()* will not add the item to the heap it returns. In this case, either manually increment the version number, reassign the value, or pass `data="all"` to *get_heap()*.

version

Version number

class spead2.**ItemGroup**

Items are collected into sets called *item groups*, which can be indexed by either item ID or item name.

There are some subtleties with respect to re-issued item descriptors. There are two cases:

1. The item descriptor is identical to a previous seen one. In this case, no action is taken.
2. Otherwise, any existing items with the same name or ID (which could be two different items) are dropped, the new item is added, and its value becomes `None`. The version is set to be higher than version on an item that was removed, so that consumers who only check the version will detect the change.

add_item(**args*, ***kwargs*)

Add a new item to the group. The parameters are used to construct an *Item*. If *id* is *None*, it will be automatically populated with an ID that is not already in use.

See the class documentation for the behaviour when the name or ID collides with an existing one. In addition, if the item descriptor is identical to an existing one and a value, this value is assigned to the existing item.

keys()

Item names

values()

Item values

items()

Dictionary style (name, value) pairs

ids()

Item IDs

update(*heap*, *new_order='='*)

Update the item descriptors and items from an incoming heap.

Parameters

- **heap** (*spead2.recv.Heap*) – Incoming heap
- **new_order** (*str*) – Byte ordering to coerce new byte arrays into. The default is to force arrays to native byte order. Use `'|'` to keep whatever byte order was in the heap. See `np.dtype.newbyteorder()` for the full list of options.

Returns

Items that have been updated from this heap, indexed by name

Return type

dict

2.5 Thread pools

The actual sending and receiving of packets is done by separate C threads. Each stream is associated with a *thread pool*, which is a pool of threads able to process its packets. See the *performance guidelines* for advice on how many threads to use.

There is one important consideration for deciding whether streams share a thread pool: if a received stream is not being consumed, it may block one of the threads from the thread pool[1]. Thus, if several streams share a thread pool, it is important to be responsive to all of them. Deciding that one stream is temporarily uninteresting and can be discarded while listening only to another one can thus lead to a deadlock if the two streams share a thread pool with only one thread.

class spead2.**ThreadPool**(*threads=1*, *affinity=[]*)

Construct a thread pool and start the threads. A list of integers can be provided for *affinity* to have the threads bound to specific CPU cores (this is only implemented for glibc). If there are fewer values than threads, the list is reused cyclically (although in this case you're probably better off having fewer threads in this case).

stop()

Shut down the worker threads. Calling this while there are still open streams is not advised. In most cases, garbage collection is sufficient.

static set_affinity(*core*)

Binds the caller to CPU core *core*.

2.6 Receiving

The classes associated with receiving are in the `spead2.recv` package. A *stream* represents a logical stream, in that packets with the same heap ID are assumed to belong to the same heap. A stream can have multiple physical transports.

Streams yield *heaps*, which are the basic units of data transfer and contain both item descriptors and item values. While it is possible to directly inspect heaps, this is not recommended or supported. Instead, heaps are normally passed to `spead2.ItemGroup.update()`.

class spead2.recv.**Heap**

cnt

Heap identifier (read-only)

flavour

SPEAD flavour used to encode the heap (see *SPEAD flavours*)

is_start_of_stream()

Returns true if the packet contains a stream start control item.

is_end_of_stream()

Returns true if the packet contains a stream stop control item.

Note: Malformed packets (such as an unsupported SPEAD version, or inconsistent heap lengths) are dropped, with a log message. However, errors in interpreting a fully assembled heap (such as invalid/unsupported formats, data of the wrong size and so on) are reported as `ValueError` exceptions. Robust code should thus be prepared to catch exceptions from heap processing.

---

[1] This is a limitation of the current design that will hopefully be overcome in future versions.

2.6.1 Configuration

Once a stream is constructed, the configuration cannot be changed. The configuration is captured in two classes, `StreamConfig` and `RingStreamConfig`. The split is a reflection of the C++ API and not particularly relevant in Python. The configuration options can either be passed to the constructors (as keyword arguments) or set as properties after construction.

class spead2.recv.**StreamConfig**(***kwargs*)

Parameters

- **max_heaps** (*int*) – The number of partial heaps that can be live at one time, per substream. This affects how intermingled heaps can be (due to out-of-order packet delivery) before heaps get dropped. See *Packet ordering* for details.
- **substreams** (*int*) – Set the number of parallel streams. The remainder when the heap cnt is divided by this value is used to identify the substream. See *Packet ordering* for details.
- **bug_compat** (*int*) – Bug compatibility flags (see *SPEAD flavours*)
- **memcpy** (*int*) – Set the method used to copy data from the network to the heap. The default is `MEMCPY_STD`. This can be changed to `MEMCPY_NONTEMPORAL`, which writes to the destination with a non-temporal cache hint (if SSE2 is enabled at compile time). This can improve performance with large heaps if the data is not going to be used immediately, by reducing cache pollution. Be careful when benchmarking: receiving heaps will generally appear faster, but it can slow down subsequent processing of the heap because it will not be cached.
- **memory_allocator** (`spead2.MemoryAllocator`) – Set the memory allocator for a stream. See *Memory allocators* for details.
- **stop_on_stop_item** (*bool*) – By default, a heap containing a stream control stop item will terminate the stream (and that heap is discarded). In some cases it is useful to keep the stream object alive and ready to receive a following stream. Setting this attribute to `False` will disable this special treatment. Such heaps can then be detected with `is_end_of_stream()`.
- **allow_unsized_heaps** (*bool*) – By default, spead2 caters for heaps without a *HEAP_LEN* item, and will dynamically extend the memory allocation as data arrives. However, this can be expensive, and ideally senders should include this item. Setting this attribute to `False` will cause packets without this item to be rejected.
- **allow_out_of_order** (*bool*) – Whether to allow packets within a heap to be received out-of-order. See *Packet ordering* for details.
- **stream_id** (*int*) – An arbitrary integer to associate with the stream. This is used to identify chunks generated by `spead2.recv.ChunkRingStream`.
- **explicit_start** (*bool*) – If set to true, the stream will not receive any data until `spead2.recv.Stream.start()` is called. See *Explicit start* for details.

Raises

ValueError – if *max_heaps* is zero.

add_stat(*name*, *mode=StreamStatConfig.COUNTER*)

Register a *custom statistic*. The return value is the index for the statistic.

Raises

ValueError – if *name* already exists

next_stat_index()

The index that will be returned by the next call to `add_stat()`.

get_stat_index (*name*)

Get the index of statistic *name*.

Raises

IndexError – if *name* is not a known statistic name

stats

Read-only list of `StreamStatConfig` describing all the statistics for the stream (including core ones). Positions in this list correspond to indices returned by *get_stat_index()*.

class spead2.recv.**RingStreamConfig** (***kwargs*)

Parameters

- **heaps** (*int*) – The capacity of the ring buffer between the network threads and the consumer. Increasing this may reduce lock contention at the cost of more memory usage.
- **contiguous_only** (*bool*) – If set to `False`, incomplete heaps will be included in the stream as instances of *IncompleteHeap*. By default they are discarded. See *Incomplete Heaps* for details.
- **incomplete_keep_payload_ranges** (*bool*) – If set to `True`, it is possible to retrieve information about which parts of the payload arrived in incomplete heaps, using `IncompleteHeap.payload_ranges()`.

Raises

ValueError – if *ring_heaps* is zero.

2.6.2 Blocking receive

To do blocking receive, create a *spead2.recv.Stream*, and add transports to it with *add_buffer_reader()*, *add_udp_reader()*, *add_tcp_reader()* or *add_udp_pcap_file_reader()*. Then either iterate over it, or repeatedly call *get()*.

class spead2.recv.**Stream** (*thread_pool*, *stream_config=StreamConfig()*, *ring_config=RingStreamConfig()*)

Parameters

- **thread_pool** (*spead2.ThreadPool*) – Thread pool handling the I/O
- **config** (*spead2.recv.StreamConfig*) – Stream configuration
- **ring_config** (*spead2.recv.RingStreamConfig*) – Ringbuffer configuration

config

Stream configuration passed to the constructor (read-only)

ring_config

Ringbuffer configuration passed to the constructor (read-only)

add_buffer_reader (*buffer*)

Feed data from an object implementing the buffer protocol.

add_udp_reader (*port*, *max_size=DEFAULT_UDP_MAX_SIZE*, *buffer_size=DEFAULT_UDP_BUFFER_SIZE*, *bind_hostname=''*, *socket=None*)

Feed data from a UDP port.

Parameters

- **port** (*int*) – UDP port number

- **max_size** (*int*) – Largest packet size that will be accepted.
- **buffer_size** (*int*) – Kernel socket buffer size. If this is 0, the OS default is used. If a buffer this large cannot be allocated, a warning will be logged, but there will not be an error.
- **bind_hostname** (*str*) – If specified, the socket will be bound to the first IP address found by resolving the given hostname. If this is a multicast group, then it will also subscribe to this multicast group.

add_udp_reader(*multicast_group*, *port*, *max_size=DEFAULT_UDP_MAX_SIZE*, *buffer_size=DEFAULT_UDP_BUFFER_SIZE*, *interface_address*)

Feed data from a UDP port (IPv4 only). This is intended for use with multicast, but it will also accept a unicast address as long as it is the same as the interface address.

Parameters

- **multicast_group** (*str*) – Hostname/IP address of the multicast group to subscribe to
- **port** (*int*) – UDP port number
- **max_size** (*int*) – Largest packet size that will be accepted.
- **buffer_size** (*int*) – Kernel socket buffer size. If this is 0, the OS default is used. If a buffer this large cannot be allocated, a warning will be logged, but there will not be an error.
- **interface_address** (*str*) – Hostname/IP address of the interface which will be subscribed, or the empty string to let the OS decide.

add_udp_reader(*multicast_group*, *port*, *max_size=DEFAULT_UDP_MAX_SIZE*, *buffer_size=DEFAULT_UDP_BUFFER_SIZE*, *interface_index*)

Feed data from a UDP port with multicast (IPv6 only).

Parameters

- **multicast_group** (*str*) – Hostname/IP address of the multicast group to subscribe to
- **port** (*int*) – UDP port number
- **max_size** (*int*) – Largest packet size that will be accepted.
- **buffer_size** (*int*) – Kernel socket buffer size. If this is 0, the OS default is used. If a buffer this large cannot be allocated, a warning will be logged, but there will not be an error.
- **interface_index** (*str*) – Index of the interface which will be subscribed, or 0 to let the OS decide.

add_tcp_reader(*port*, *max_size=DEFAULT_TCP_MAX_SIZE*, *buffer_size=DEFAULT_TCP_BUFFER_SIZE*, *bind_hostname=''*)

Receive data over TCP/IP. This will listen for a single incoming connection, after which no new connections will be accepted. When the connection is closed, the stream is stopped.

Parameters

- **port** (*int*) – TCP port number
- **max_size** (*int*) – Largest packet size that will be accepted.
- **buffer_size** (*int*) – Kernel socket buffer size. If this is 0, the OS default is used. If a buffer this large cannot be allocated, a warning will be logged, but there will not be an error.
- **bind_hostname** (*str*) – If specified, the socket will be bound to the first IP address found by resolving the given hostname.

add_tcp_reader(*acceptor*, *max_size=DEFAULT_TCP_MAX_SIZE*)

Receive data over TCP/IP. This is similar to the previous overload, but takes a user-provided socket, which must already be listening for connections. It duplicates the acceptor socket, so the original can be closed immediately.

Parameters

- **acceptor** (*socket.socket*) – Listening socket
- **max_size** (*int*) – Largest packet size that will be accepted.

add_udp_pcap_file_reader(*filename*, *filter=''*)

Feed data from a pcap file (for example, captured with **tcpdump** or *mcdump*). An optional filter selects a subset of the packets from the capture file. This is only available if libpcap development files were found at compile time.

Parameters

- **filename** (*str*) – Filename of the capture file
- **filter** (*str*) – Filter to apply to packets from the capture file

add_inproc_reader(*queue*)

Feed data from an in-process queue. Refer to *In-process transport* for details.

get()

Returns the next heap, blocking if necessary. If the stream has been stopped, either by calling *stop()* or by receiving a stream control packet, it raises `spead2.Stopped`. However, heap that were already queued when the stream was stopped are returned first.

A stream can also be iterated over to yield all heaps.

get_nowait()

Like *get()*, but if there is no heap available it raises `spead2.Empty`.

start()

Start receiving data. This only needs to be called if the *explicit_start* parameter to *spead2.recv.StreamConfig* is set to `True`, although it is harmless to call it even if not. If *explicit_start* is true, then after calling *start()*, it is no longer possible to add more readers.

stop()

Shut down the stream and close all associated sockets. It is not possible to restart a stream once it has been stopped; instead, create a new stream.

fd

The read end of a pipe to which a byte is written when a heap is received. **Do not read from this pipe.** It is used for integration with asynchronous I/O frameworks (see below).

stats

Statistics about the stream.

ringbuffer

The internal ringbuffer of the stream (see *Statistics*).

2.6.3 Asynchronous receive

Asynchronous I/O is supported through Python's `asyncio` module. It can be combined with other asynchronous I/O frameworks like twisted and Tornado.

class spead2.recv.asyncio.**Stream**(**args*, ***kwargs*)

See *spead2.recv.Stream* (the base class) for other constructor arguments.

get()

Coroutine that yields the next heap, or raises `spead2.Stopped` once the stream has been stopped and there is no more data. It is safe to have multiple in-flight calls, which will be satisfied in the order they were made.

The stream is also asynchronously iterable, i.e., can be used in an `async for` loop to iterate over the heaps.

2.6.4 Packet ordering

SPEAD is typically carried over UDP, and by its nature, UDP allows packets to be reordered. Packets may also arrive interleaved if they are produced by multiple senders. We consider two sorts of packet ordering issues:

1. Re-ordering within a heap. By default, spead2 assumes that all the packets that form a heap will arrive in order, and discards any packet that does not have the expected payload offset. In most networks this is a safe assumption provided that all the packets originate from the same sender (IP address and port number) and have the same destination.

   If this assumption is not appropriate, it can be changed with the `allow_out_of_order` attribute of *spead2.recv.StreamConfig*. This has minimal impact when packets do in fact arrive in order, but reassembling arbitrarily ordered packets can be expensive. Allowing for out-of-order arrival also makes handling lost packets more expensive (because one must cater for them arriving later), which can lead to a feedback loop as this more expensive processing can lead to further packet loss.

2. Interleaving of packets from different heaps. This is always supported, but to a bounded degree so that lost packets don't lead to heaps being kept around indefinitely in the hope that the packet may arrive. The `max_heaps` attribute of *spead2.recv.StreamConfig* determines the amount of overlap allowed: once a packet in heap n is observed, it is assumed that heap $n - \text{max_heaps}$ is complete. When there are many producers it will likely be necessary to increase this value. Larger values increase the memory usage for partial heaps, and have a small performance impact.

   It's possible to get more predictable results when the producers interleave their heap cnts (for example, by using `spead2.send.Stream.set_cnt_sequence()`) such that the remainder when dividing the heap cnt by the number of producers identifies the producer. In this case, set the `substreams` attribute of *spead2.recv.StreamConfig* to the number of producers. Note that `max_heaps` applies separately to each producer, and can usually be very low (1 or 2) if the producer sends one heap at a time.

2.6.5 Memory allocators

To allow for performance tuning, it is possible to use an alternative memory allocator for heap payloads. A few allocator classes are provided; new classes must currently be written in C++. The default (which is also the base class for all allocators) is `spead2.MemoryAllocator`, which has no constructor arguments or methods. An alternative is *spead2.MmapAllocator*.

class spead2.**MmapAllocator**(*flags=0*, *prefer_huge=False*)

An allocator using *mmap(2)*. This may be slightly faster for large allocations, and allows setting custom mmap flags. This is mainly intended for use with the C++ API, but is exposed to Python as well.

Parameters

- **flags** (*int*) – Extra flags to pass to *mmap(2)*. Finding the numeric values for OS-specific flags is left as a problem for the user.
- **prefer_huge** (*bool*) – If true, allocations will try to use huge pages (if supported by the OS), and fall back to normal pages if that fails.

The most important custom allocator is *spead2.MemoryPool*. It allocates from a pool, rather than directly from the system. This can lead to significant performance improvements when the allocations are large enough that the C library allocator does not recycle the memory itself, but instead requests memory from the kernel.

A memory pool has a range of sizes that it will handle from its pool, by allocating the upper bound size. Thus, setting too wide a range will waste memory, while setting too narrow a range will prevent the memory pool from being used at all. A memory pool is best suited for cases where the heaps are all roughly the same size.

A memory pool can optionally use a background task (scheduled onto a thread pool) to replenish the pool when it gets low. This is useful when heaps are being captured and stored indefinitely rather than processed and released.

class spead2.**MemoryPool**(*thread_pool*, *lower*, *upper*, *max_free*, *initial*, *low_water*, *allocator=None*)

Constructor. One can omit *thread_pool* and *low_water* to skip the background refilling.

Parameters

- **thread_pool** (ThreadPool) – thread pool used for refilling the memory pool
- **lower** (*int*) – Minimum allocation size to handle with the pool
- **upper** (*int*) – Size of allocations to make
- **max_free** (*int*) – Maximum number of allocations held in the pool
- **initial** (*int*) – Number of allocations to put in the free pool initially.
- **low_water** (*int*) – When fewer than this many buffers remain, the background task will be started and allocate new memory until *initial* buffers are available.
- **allocator** (*MemoryAllocator*) – Underlying memory allocator

warn_on_empty

Whether to issue a warning if the memory pool becomes empty and needs to allocate new memory on request. It defaults to true.

2.6.6 Incomplete Heaps

By default, an incomplete heap (one for which some but not all of the packets were received) is simply dropped and a warning is printed. Advanced users might need finer control, such as recording metrics about the number of these heaps. To do so, set *contiguous_only* to `False` in the *RingStreamConfig*. The stream will then yield instances of *IncompleteHeap*.

class spead2.recv.**IncompleteHeap**

cnt

Heap identifier (read-only)

flavour

SPEAD flavour used to encode the heap (see *SPEAD flavours*)

heap_length

The expected number of bytes of payload (-1 if unknown)

received_length

The number of bytes of payload that were actually received

payload_ranges

A list of pairs of heap offsets. Each pair is a range of bytes that was received. This is only non-empty if *incomplete_keep_payload_ranges* was set in the `RingStreamConfig`; otherwise the information is dropped to save memory.

When using this, you should also set *allow_out_of_order* to `True` in the `StreamConfig`, as otherwise any data after the first lost packet is discarded.

is_start_of_stream()

Returns true if the packet contains a stream start control item.

is_end_of_stream()

Returns true if the packet contains a stream stop control item.

2.6.7 Statistics

Refer to *Receiver stream statistics* for general information about statistics.

class spead2.recv.**StreamStats**

Collection of statistics. It present both dictionary-like and sequence-like interfaces. Iteration is dictionary-like, iterating over the keys (names of statistics). Indexing with negative indices is not supported.

config

List of `spead2.recv.StreamStatConfig` describing the available statistics in further detail. This gives the same list as `StreamConfig.stats`.

class spead2.recv.**StreamStatConfig**

class Mode

COUNTER

MAXIMUM

name: str

Name of the statistic

mode: *Mode*

Mode for updating long-term statistics from per-batch statistics

combine(*a*, *b*)

Combine two samples according to the mode.

Additional statistics are available on the ringbuffer underlying the stream (`ringbuffer` property), with similar caveats about synchronisation.

class spead2.recv.Stream.**Ringbuffer**

size()

Number of heaps currently in the ringbuffer.

capacity()

Maximum number of heaps that can be held in the ringbuffer (corresponds to the *heaps* attribute of `RingStreamConfig`).

The `spead2.recv.stream_stat_indices` module contains constants for indices that can be used to retrieve core statistics by index, without needing to look up the index.

spead2.recv.stream_stat_indices.**HEAPS**

spead2.recv.stream_stat_indices.**INCOMPLETE_HEAPS_EVICTED**

spead2.recv.stream_stat_indices.**INCOMPLETE_HEAPS_FLUSHED**

spead2.recv.stream_stat_indices.**PACKETS**

spead2.recv.stream_stat_indices.**BATCHES**

spead2.recv.stream_stat_indices.**MAX_BATCH**

spead2.recv.stream_stat_indices.**SINGLE_PACKET_HEAPS**

spead2.recv.stream_stat_indices.**SEARCH_DIST**

spead2.recv.stream_stat_indices.**WORKER_BLOCKED**

2.6.8 Explicit start

When using multiple readers with a stream or multiple streams, it is sometimes desirable to have them all begin listening to the network at the same time, so that their data can be matched up. Adding readers can be slow (mostly due to the cost of allocating buffers), so when adding multiple readers serially, they will start listening at very different times.

If one sets the *explicit_start* parameter to `spead2.recv.StreamConfig` to true, then adding a reader will do the expensive work of allocating buffers, but will not start it listening to the network. That is done by calling `spead2.recv.Stream.start()`. This will still iterate serially over the readers, so they will not start listening at exactly the same time, but the skew will be much smaller because the operation is much more light-weight.

When this feature is in use, no readers can be added to a stream after calling `start()` (doing so will raise an exception). This gives the implementation a hint that adding readers cannot happen concurrently with packets arriving. At present the implementation does not take advantage of this assumption, but that is subject to change in future versions of spead2.

2.7 Sending

Unlike for receiving, each stream object can only use a single transport. There is currently no support for collective operations where multiple producers cooperate to construct a heap between them. It is still possible to do multi-producer, single-consumer operation if the heap IDs are kept separate.

Because each stream has only one transport, there is a separate class for each, rather than a generic *Stream* class. Because there is common configuration between the stream classes, configuration is encapsulated in a `spead2.send.StreamConfig`.

class spead2.send.**StreamConfig**(**, max_packet_size=1472, rate=0.0, burst_size=65536, max_heaps=4, burst_rate_ratio=1.05*)

Parameters

- **max_packet_size** (`int`) – Heaps will be split into packets of at most this size.

- **rate** (*double*) – Target transmission rate, in bytes per second, or 0 to send as fast as possible.
- **burst_size** (*int*) – Bursts of up to this size will be sent as fast as possible. Setting this too large (larger than available buffer sizes) risks losing packets, while setting it too small may reduce throughput by causing more sleeps than necessary.
- **max_heaps** (*int*) – For asynchronous transmits, the maximum number of heaps that can be in-flight.
- **burst_rate_ratio** (*float*) – If packet sending falls below the target transmission rate, the rate will be increased until the average rate has caught up. This value specifies the "catch-up" rate, as a ratio to the target rate.
- **rate_method** (RateMethod) – Select method for applying the rate limit. If true, then hardware-based rate limiting may be used if available. In this case it is implementation-defined whether *burst_rate_ratio* and *burst_size* have any effect. This will often produce results that are at least as good as the software limiter, but in some cases (particularly higher data rates) the overall rate becomes less accurate and so it is disabled by default.

The constructor arguments are also instance attributes.

class spead2.send.**RateMethod**

An enumeration to select a method for rate limiting.

SW

Use a generic software rate limiter.

HW

Use a hardware rate limiter, if available. This is currently only supported when using *ibverbs*, and only if the hardware supports it. If hardware rate limiting is not available, falls back to software.

AUTO

This is the default, and lets the implementation decide whether to use hardware rate limiting. At present it will never use the hardware rate limiter (because the available hardware limiters don't do a good job under all conditions), but in future it is likely to use the hardware limiter more often in circumstances where it has been tested to perform well.

Streams send pre-baked heaps, which can be constructed by hand, but are more normally created from an *ItemGroup* by a *spead2.send.HeapGenerator*. To simplify cases where one item group is paired with one heap generator, a convenience class spead2.send.ItemGroup is provided that inherits from both.

class spead2.send.**HeapGenerator**(*item_group*, *descriptor_frequency=None*, *flavour=<spead2._spead2.Flavour object>*)

Tracks which items and item values have previously been sent and generates delta heaps.

Parameters

- **item_group** (*spead2.ItemGroup*) – Item group to monitor.
- **descriptor_frequency** (*int, optional*) – If specified, descriptors will be re-sent once every *descriptor_frequency* heaps generated by this method.
- **flavour** (spead2.Flavour) – The SPEAD protocol flavour used for heaps generated by *get_heap()* and *get_end()*.

add_to_heap(*heap*, *descriptors='stale'*, *data='stale'*)

Update a heap to contains all the new items and item descriptors since the last call.

Parameters

- **heap** (*Heap*) – The heap to update.

- **descriptors** (*{'stale', 'all', 'none'}*) – Which descriptors to send. The default ('stale') sends only descriptors that have not been sent, or have not been sent recently enough according to the *descriptor_frequency* passed to the constructor. The other options are to send all the descriptors or none of them. Sending all descriptors is useful if a new receiver is added which will be out of date.
- **data** (*{'stale', 'all', 'none'}*) – Which data items to send.
- **item_group** (ItemGroup, optional) – If specified, uses the items from this item group instead of the one passed to the constructor (which could be *None*).

Raises

ValueError – if *descriptors* or *data* is not one of the legal values

get_heap(**args*, ***kwargs*)

Return a new heap which contains all the new items and item descriptors since the last call. This is a convenience wrapper around add_to_heap().

get_start()

Return a heap that contains only a start-of-stream marker.

get_end()

Return a heap that contains only an end-of-stream marker.

class spead2.send.**Heap**(*flavour=spead2.Flavour()*)

repeat_pointers

Enable/disable repetition of item pointers in all packets.

Usually this is not needed, but it can enable some specialised use cases where immediates can be recovered from incomplete heaps or where the receiver examines the item pointers in each packet to decide how to handle it. The packet size must be large enough to fit all the item pointers for the heap (the implementation also reserves a little space, so do not rely on a tight fit working).

The default is disabled.

add_item(*item*)

Add an Item to the heap. This references the memory in the item rather than copying it. It does *not* cause a descriptor to be sent; use add_descriptor() for that.

add_descriptor(*descriptor*)

Add a Descriptor to the heap.

add_start()

Convenience method to add a start-of-stream item.

add_end()

Convenience method to add an end-of-stream item.

2.7.1 Substreams

For some transport types it is possible to create a stream with multiple "substreams". Each substream typically has a separate destination address, but all the heaps within the stream are sent in order, and the stream configuration (including the rate limits) applies to the stream as a whole. Using substreams rather than independent streams gives better control over the overall transmission rate, and uses fewer system resources.

When sending a heap, an optional parameter called *substream_index* selects the substream that will be used.

2.7.2 Blocking send

There are multiple stream classes, corresponding to different transports, and some of the classes have several variants of the constructor. They all implement the following interface (the class exists as a type annotation, but does not currently exist at runtime).

class spead2.send.**SyncStream**

send_heap(*heap*, *cnt=-1*, *substream_index=0*, *rate=-1.0*)

Send a *spead2.send.Heap* to the peer, and wait for completion. There is currently no indication of whether it successfully arrived, but `IOError` is raised if it could not be sent.

If not specified, a heap cnt is chosen automatically (the choice can be modified by calling *set_cnt_sequence()*). If a non-negative value is specified for *cnt*, it is used instead. It is the user's responsibility to avoid collisions.

See *Substreams* for a description of *substream_index*.

Normally a rate is set for the whole stream, but it can be overridden here by providing a non-negative value. See *spead2.send.StreamConfig* for the interpretation.

send_heaps(*heaps*, *mode*)

Send a group of heaps. See *Batching* for more information.

This function will either enqueue all of the heaps, or none of them. In particular, there must be space in the queue for all of them.

It is an error for *heaps* to be empty. Currently this raises `OSError`, but it may be replaced by `ValueError` in future.

Parameters

- **heaps** (*List*[`spead2.send.HeapReference`] | `spead2.send.HeapReferenceList`) – A list of heaps to send
- **mode** (`spead2.send.GroupMode`) – Controls the packet ordering

set_cnt_sequence(*next*, *step*)

Modify the linear sequence used to generate heap cnts. The next heap will have cnt *next*, and each following cnt will be incremented by *step*. When using this, it is the user's responsibility to ensure that the generated values remain unique. The initial state is *next* = 1, *step* = 1.

This is useful when multiple senders will send heaps to the same receiver, and need to keep their heap cnts separate.

If the computed cnt overflows the number of bits available, the bottom-most bits are taken.

num_substreams

Number of substreams in this stream (read-only).

UDP

Note that since UDP is an unreliable protocol, there is no guarantee that packets arrive.

For each constructor overload, the *endpoints* parameter can also be replaced by two parameters that contain the hostname/IP address and port for a single substream (for backwards compatibility).

class spead2.send.**UdpStream**(*thread_pool*, *endpoints*, *config=spead2.send.StreamConfig()*, *buffer_size=DEFAULT_BUFFER_SIZE*, *interface_address=''*)

Parameters

- **thread_pool** (*spead2.ThreadPool*) – Thread pool handling the I/O
- **endpoints** (*List[Tuple[str, int]]*) – Peer endpoints (one per substream)
- **config** (*spead2.send.StreamConfig*) – Stream configuration
- **buffer_size** (*int*) – Socket buffer size. A warning is logged if this size cannot be set due to OS limits.
- **interface_address** (*str*) – Source hostname/IP address (see tips about *Routing*).

class spead2.send.**UdpStream**(*thread_pool*, *endpoints*, *config=spead2.send.StreamConfig()*, *buffer_size=DEFAULT_BUFFER_SIZE*, *ttl*)

Stream using UDP, with multicast TTL. Note that the regular constructor will also work with multicast, but does not give any control over the TTL.

Parameters

- **thread_pool** (*spead2.ThreadPool*) – Thread pool handling the I/O
- **endpoints** (*List[Tuple[str, int]]*) – Peer endpoints (one per substream)
- **config** (*spead2.send.StreamConfig*) – Stream configuration
- **buffer_size** (*int*) – Socket buffer size. A warning is logged if this size cannot be set due to OS limits.
- **ttl** (*int*) – Multicast TTL

class spead2.send.**UdpStream**(*thread_pool*, *endpoints*, *config=spead2.send.StreamConfig()*, *buffer_size=524288*, *ttl*, *interface_address*)

Stream using UDP, with multicast TTL and interface address (IPv4 only).

Parameters

- **thread_pool** (*spead2.ThreadPool*) – Thread pool handling the I/O
- **endpoints** (*List[Tuple[str, int]]*) – Peer endpoints (one per substream)
- **config** (*spead2.send.StreamConfig*) – Stream configuration
- **buffer_size** (*int*) – Socket buffer size. A warning is logged if this size cannot be set due to OS limits.
- **ttl** (*int*) – Multicast TTL
- **interface_address** (*str*) – Hostname/IP address of the interface on which to send the data

class spead2.send.**UdpStream**(*thread_pool*, *endpoints*, *config=spead2.send.StreamConfig()*, *buffer_size=DEFAULT_BUFFER_SIZE*, *ttl*, *interface_index*)

Stream using UDP, with multicast TTL and interface index (IPv6 only).

Parameters

- **thread_pool** (*spead2.ThreadPool*) – Thread pool handling the I/O
- **endpoints** (*List[Tuple[str, int]]*) – Peer endpoints (one per substream)
- **config** (*spead2.send.StreamConfig*) – Stream configuration
- **buffer_size** (*int*) – Socket buffer size. A warning is logged if this size cannot be set due to OS limits.
- **ttl** (*int*) – Multicast TTL
- **interface_index** (*str*) – Index of the interface on which to send the data

class spead2.send.**UdpStream**(*thread_pool*, *socket*, *endpoints*, *config=spead2.send.StreamConfig()*)

Stream using UDP, with a pre-existing socket. The socket is duplicated by the stream, so the original can be closed immediately to free up a file descriptor. The caller is responsible for setting any socket options. The socket must not be connected.

Parameters

- **thread_pool** (*spead2.ThreadPool*) – Thread pool handling the I/O
- **socket** (*socket.socket*) – UDP socket
- **endpoints** (*List[Tuple[str, int]]*) – Peer endpoints (one per substream)
- **config** (*spead2.send.StreamConfig*) – Stream configuration

class spead2.send.**UdpStream**(*thread_pool*, *endpoints*, *config=spead2.send.StreamConfig()*, *buffer_size=DEFAULT_BUFFER_SIZE*, *socket*)

Parameters

- **thread_pool** (*spead2.ThreadPool*) – Thread pool handling the I/O
- **endpoints** (*List[Tuple[str, int]]*) – Peer endpoints (one per substream)
- **config** (*spead2.send.StreamConfig*) – Stream configuration
- **buffer_size** (*int*) – Socket buffer size. A warning is logged if this size cannot be set due to OS limits.

TCP

TCP/IP is a reliable protocol, so heap delivery is guaranteed. However, if multiple threads all call `send_heap()` at the same time, they can exceed the configured *max_heaps* and heaps will be dropped.

Because spead2 was originally designed for UDP, the default packet size in `StreamConfig` is quite small. Performance can be improved by increasing it (but be sure the receiver is configured to handle larger packets).

TCP/IP is also a connection-oriented protocol, and does not support substreams. The *endpoints* must therefore contain exactly one endpoint (it takes a list for consistency with `UdpStream`).

class spead2.send.**TcpStream**(*thread_pool*, *endpoints*, *config=spead2.send.StreamConfig()*, *buffer_size=DEFAULT_BUFFER_SIZE*, *interface_address=''*)

Parameters

- **thread_pool** (*spead2.ThreadPool*) – Thread pool handling the I/O
- **endpoints** (*List[Tuple[str, int]]*) – Peer endpoint (must contain exactly one element).

- **config** (*spead2.send.StreamConfig*) – Stream configuration
- **buffer_size** (*int*) – Socket buffer size. A warning is logged if this size cannot be set due to OS limits.
- **interface_address** (*str*) – Source hostname/IP address (see tips about *Routing*).

class spead2.send.**TcpStream**(*thread_pool*, *socket*, *config=spead2.send.StreamConfig()*)

Stream using an existing socket. The socket must already be connected to the peer, and the user is responsible for setting any desired socket options. The socket is duplicated, so it can be closed to save resources.

Parameters

- **thread_pool** (*spead2.ThreadPool*) – Thread pool handling the I/O
- **socket** (*socket.socket*) – TCP socket
- **config** (*spead2.send.StreamConfig*) – Stream configuration

Raw bytes

class spead2.send.**BytesStream**(*thread_pool*, *config=spead2.send.StreamConfig()*)

Stream that collects packets in memory and makes the concatenated stream available.

Parameters

- **thread_pool** (*spead2.ThreadPool*) – Thread pool handling the I/O
- **config** (*spead2.send.StreamConfig*) – Stream configuration

getvalue()

Return a copy of the memory buffer.

Return type

bytes

In-process transport

Refer to the separate *documentation*.

2.7.3 Asynchronous send

As for asynchronous receives, asynchronous sends are managed by asyncio. A stream can buffer up multiple heaps for asynchronous send, up to the limit specified by *max_heaps* in the `StreamConfig`. If this limit is exceeded, heaps will be dropped, and the returned future has an `IOError` exception set. An `IOError` could also indicate a low-level error in sending the heap (for example, if the packet size exceeds the MTU).

The classes exist in the `spead2.send.asyncio` modules, and mostly implement the same constructors as the synchronous classes. They implement the following interface (the class exists as a type annotation, but does not currently exist at runtime):

class spead2.send.asyncio.**AsyncStream**

async_send_heap(*heap*, *cnt=-1*, *substream_index=0*, *rate=-1.0*)

Send a heap asynchronously. Note that this is *not* a coroutine: it returns a future. Adding the heap to the queue is done synchronously, to ensure proper ordering.

Parameters

- **heap** (*spead2.send.Heap*) – Heap to send
- **cnt** (*int*) – Heap cnt to send (defaults to auto-incrementing)
- **rate** (*float*) – Rate at which to transmit (defaults to the stream's rate)

async_send_heaps(*heaps*, *mode*)

Send a group of heaps asynchronously. See *Batching* for more information. Note that this is *not* a coroutine: it returns a future. Adding the heaps to the queue is done synchronously, to ensure proper ordering.

The parameters have the same meaning as for `send_heaps()`.

Parameters

- **heaps** (*List[spead2.send.HeapReference] | spead2.send.HeapReferenceList*) – A list of heaps to send
- **mode** (*spead2.send.GroupMode*) – Controls the packet ordering

flush()

Block until all enqueued heaps have been sent (or dropped).

async_flush()

Asynchronously wait for all enqueued heaps to be sent. Note that this only waits for heaps passed to `async_send_heap()` prior to this call, not ones added while waiting.

TCP

For TCP, construction is slightly different: except when providing a custom socket, one uses a coroutine to connect:

async classmethod TcpStream.**connect**(**args*, ***kwargs*)

Open a connection.

The arguments are the same as for the constructor of `spead2.send.TcpStream`.

2.7.4 Batching

Instead of sending one heap at a time, it is possible to pass a whole list of heaps to be sent at once. There are a few reasons one might want to do this:

1. It is generally more efficient, particularly if the heaps are small.
2. The packets of the heaps can be sent in an interleaved order. This is useful when combined with *Substreams*, as each substream can have a steady flow of packets rather than sending a full heap to one substream, then a full heap to the next etc.

class spead2.send.**HeapReference**(*heap*, ***, *cnt=-1*, *substream_index=0*, *rate=-1.0*)

A thin wrapper around a `Heap`, heap cnt and substream index, for passing to `send_heaps()`. The parameters have the same meaning as the corresponding arguments to `send_heap()`.

class spead2.send.**GroupMode**

Enumeration selecting the packet ordering for a group of heaps sent with `send_heaps()`.

ROUND_ROBIN

Interleave the packets of the heaps. One packet is sent from each heap in turn (skipping those that have run out of packets).

SERIAL
Send the heaps one after another.

Passing a large list has some overhead as the list has to be converted from Python to C++. If exactly the same list will be used multiple times, this cost can be amortised by converting the list to a `HeapReferenceList` up front and then using repeatedly.

class `spead2.send.HeapReferenceList(heaps)`
An opaque copy of a list of `HeapReference`. It can be passed to `send_heaps()` in place of a list. It can also be indexed with a slice to create a new `HeapReferenceList` with a subset of the original heaps.

Parameters
heaps (`List[spead2.send.HeapReference]`) – The heap references to store

2.8 In-process transport

While SPEAD is generally deployed over UDP, it is less than ideal for writing tests:

- One has to deal with allocating port numbers and avoiding conflicts.
- The sender and receiver need to be running at the same time.
- If the receiver doesn't keep up, it can drop packets.

To simplify unit testing, spead2 also offers an "in-process" transport. One creates a queue, then connects a sender and a receiver to it. The queue has unbounded capacity, so one can safely send all the data first, then create the receiver later. This unbounded capacity also means that it should *not* be used in production for high-volume streams, because it can exhaust all your memory if the sender works faster than the receiver.

A queue can also be connected to multiple senders. It should generally not be connected to multiple receivers, because they will each get some undefined subset of the packets, which won't reassemble into the proper heaps. However, if you set the packet size large enough that every heap is contained in one packet then it will work.

> **Warning:** Even though the transport is reliable, a stream has a maximum number of outstanding heaps. Attempting to send more heaps in parallel than the stream is configured to handle can lead to heaps being dropped. This is not a problem when using a single thread with `spead2.send.InprocStream.send_heap()` (because it blocks until the heap has been fully added to the queue), but needs to be considered when sending heaps in parallel with `spead2.send.asyncio.InprocStream` or when using multiple threads.

2.8.1 Sending

class `spead2.InprocQueue`

add_packet(*packet*)
Add a packet directly to the queue.

stop()
Indicate end-of-stream to receivers. It is an error to add any more packets after this.

class `spead2.send.InprocStream(thread_pool, queues, config)`

Parameters
- **thread_pool** (`spead2.ThreadPool`) – Thread pool handling the I/O

- **queues** (List[*spead2.InprocQueue*]) – Queues holding the generated packets (one per substream).
- **config** (*spead2.send.StreamConfig*) – Stream configuration

queues

Get the queues passed to the constructor.

class spead2.send.asyncio.**InprocStream**(*thread_pool*, *queues*, *config*)

SPEAD over reliable in-process transport.

Note: Data may still be lost if the maximum number of in-flight heaps (set in the stream config) is exceeded. Either set this value to more heaps than will ever be sent (which will use unbounded memory) or be sure to block on the futures returned before exceeding the capacity.

Parameters

- **thread_pool** (*spead2.ThreadPool*) – Thread pool handling the I/O
- **queues** (List[*spead2.InprocQueue*]) – Queue holding the data in flight
- **config** (*spead2.send.StreamConfig*) – Stream configuration

An asynchronous version of *spead2.send.InprocStream*. Refer to *Asynchronous send* for general details about asynchronous transport.

2.8.2 Receiving

To connect a receiver to the the queue, use *spead2.recv.Stream.add_inproc_reader()*.

2.9 Logging

Logging is done with the standard Python `logging` module, and logging can be configured with the usual utilities. However, in the default build the debug logging is completely disabled for performance reasons. To enable it, you need to set the Meson option `max_log_level=debug`. For example, if installing with **pip**, use

```
pip install --config-settings=setup-args=-Dmax_log_level=debug .
```

2.10 Support for ibverbs

Receiver performance can be significantly improved by using the Infiniband Verbs API instead of the BSD sockets API. This is currently only tested on Linux with ConnectX® NICs. It depends on device managed flow steering (DMFS).

There are a number of limitations in the current implementation:

- Only IPv4 is supported.
- VLAN tagging, IP optional headers, and IP fragmentation are not supported.
- For sending, only multicast is supported.

Within these limitations, it is quite easy to take advantage of this faster code path. The main difficulties are that one *must* specify the IP address of the interface that will send or receive the packets, and that the `CAP_NET_RAW` capability may be needed. The netifaces2 module can help find the IP address for an interface by name, and the *spead2_net_raw* tool simplifies the process of getting the `CAP_NET_RAW` capability.

2.10.1 System configuration

ConnectX®-3

Add the following to `/etc/modprobe.d/mlnx.conf`:

```
options ib_uverbs disable_raw_qp_enforcement=1
options mlx4_core fast_drop=1
options mlx4_core log_num_mgm_entry_size=-1
```

Note: Setting `log_num_mgm_entry_size` to -7 instead of -1 will activate faster static device-managed flow steering. This has some limitations (refer to the manual for details), but can improve performance when capturing a large number of multicast groups.

ConnectX®-4+, MLNX OFED up to 4.9

Add the following to `/etc/modprobe.d/mlnx.conf`:

```
options ib_uverbs disable_raw_qp_enforcement=1
```

All other cases

No system configuration is needed, but the `CAP_NET_RAW` capability is required. Running as root will achieve this; a full discussion of Linux capabilities is beyond the scope of this manual. The *spead2_net_raw* utility can also be used to give users access to this capability without exposing full root access. For more information, see the libvma documentation.

Multicast loopback

By default, multicast traffic sent using ibverbs can also be received on the same port. While convenient, this is a slow path in the NIC, and can limit performance. To disable this loopback, write `1` to `/sys/class/net/`*`interface`*`/settings/force_local_lb_disable` (note that the setting does not persist across reboots).

2.10.2 Receiving

The ibverbs API can be used programmatically by using an extra method of `spead2.recv.Stream`.

The configuration is specified using a `spead.recv.UdpIbvConfig`.

class `spead2.recv.`**`UdpIbvConfig`**(**, endpoints=[], interface_address='', buffer_size=DEFAULT_BUFFER_SIZE, max_size=DEFAULT_MAX_SIZE, comp_vector=0, max_poll=DEFAULT_MAX_POLL*)

Parameters

- **endpoints** (*List[Tuple[str, int]]*) – Peer endpoints
- **interface_address** (*str*) – Hostname/IP address of the interface which will be subscribed
- **buffer_size** (*int*) – Requested memory allocation for work requests. It may be adjusted to an integer number of packets.
- **max_size** (*int*) – Maximum packet size that will be accepted
- **comp_vector** (*int*) – Completion channel vector (interrupt) for asynchronous operation, or a negative value to poll continuously. Polling should not be used if there are other users of the thread pool. If a non-negative value is provided, it is taken modulo the number of available completion vectors. This allows a number of streams to be assigned sequential completion vectors and have them load-balanced, without concern for the number available.
- **max_poll** (*int*) – Maximum number of times to poll in a row, without waiting for an interrupt (if *comp_vector* is non-negative) or letting other code run on the thread (if *comp_vector* is negative).

The constructor arguments are also instance attributes. Note that they are implemented as properties that return copies of the state, which means that mutating *endpoints* (for example, with `append()`) will not have any effect as only the copy will be modified. The entire list must be assigned to update it.

`spead2.recv.Stream.`**`add_udp_ibv_reader`**(*config*)

Feed data from IPv4 traffic.

If supported by the NIC and the drivers, the receive code will automatically use a "multi-packet receive queue", which allows each packet to consume only the amount of space needed in the buffer. This is currently only supported on ConnectX®-4+ with MLNX OFED drivers 5.0 or later (or upstream rdma-core). When in use, the *max_size* parameter has little impact on performance, and is used only to reject larger packets.

When multi-packet receive queues are not supported, performance can be improved by making *max_size* as small as possible for the intended data stream. This will increase the number of packets that can be buffered (because the buffer is divided into fixed-size slots), and also improve memory efficiency by keeping data more-or-less contiguous.

Environment variables

An existing application can be forced to use ibverbs for all IPv4 readers, by setting the environment variable `SPEAD2_IBV_INTERFACE` to the IP address of the interface to receive the packets. Note that calls to `spead2.recv.Stream.add_udp_reader()` that pass an explicit interface will use that interface, overriding `SPEAD2_IBV_INTERFACE`; in this case, `SPEAD2_IBV_INTERFACE` serves only to enable the override.

It is also possible to specify `SPEAD2_IBV_COMP_VECTOR` to override the completion channel vector from the default.

Note that this environment variable currently has no effect on senders.

2.10.3 Sending

Sending is done by using the class `spead2.send.UdpIbvStream` instead of `spead2.send.UdpStream`. It has a different constructor, but the same methods. There is also a `spead2.send.asyncio.UdpIbvStream` class, analogous to `spead2.send.asyncio.UdpStream`.

There is an additional configuration class for ibverbs-specific configuration:

class `spead2.send.`**`UdpIbvConfig`**(**, endpoints=[], interface_address='', buffer_size=DEFAULT_BUFFER_SIZE, ttl=1, comp_vector=0, max_poll=DEFAULT_MAX_POLL, memory_regions=[]*)

Parameters

- **endpoints** (*List[Tuple[str, int]]*) – Peer endpoints (one per substream)
- **interface_address** (*str*) – Hostname/IP address of the interface which will be subscribed
- **buffer_size** (*int*) – Requested memory allocation for work requests. It may be adjusted to an integer number of packets.
- **ttl** (*int*) – Multicast TTL
- **comp_vector** (*int*) – Completion channel vector (interrupt) for asynchronous operation, or a negative value to poll continuously. Polling should not be used if there are other users of the thread pool. If a non-negative value is provided, it is taken modulo the number of available completion vectors. This allows a number of streams to be assigned sequential completion vectors and have them load-balanced, without concern for the number available.
- **max_poll** (*int*) – Maximum number of times to poll in a row, without waiting for an interrupt (if *comp_vector* is non-negative) or letting other code run on the thread (if *comp_vector* is negative).
- **memory_regions** (*List[object]*) – Objects implementing the buffer protocol that will be used to hold item data. This is not required, but data stored in these buffers may be transmitted directly without requiring a copy, yielding higher performance. There may be platform-specific limitations on the size and number of these buffers.

The constructor arguments are also instance attributes. Note that they are implemented as properties that return copies of the state, which means that mutating *endpoints* or *memory_regions* (for example, with `append()`) will not have any effect as only the copy will be modified. The entire list must be assigned to update it.

class `spead2.send.`**`UdpIbvStream`**(*thread_pool*, *config*, *udp_ibv_config*)

Create a multicast IPv4 UDP stream using the ibverbs API

Parameters

- **thread_pool** (*spead2.ThreadPool*) – Thread pool handling the I/O
- **config** (*spead2.send.StreamConfig*) – Stream configuration
- **udp_ibv_config** (*spead2.send.UdpIbvConfig*) – Additional stream configuration

2.11 Chunking receiver

For an overview, refer to *Chunking receiver*. This page is a reference for the Python API.

2.11.1 Writing a place callback

A callback is needed to determine which chunk each heap belongs to and where it fits into that chunk. The callback is made from a C++ thread, so it cannot be written in pure Python, or even use the Python C API. It needs to be compiled code. The callback code should also not attempt to acquire the Global Interpreter Lock (GIL), as it may lead to a deadlock.

Once you've written the function (see below), it needs to be passed to spead2. The easiest way to do this is with `scipy.LowLevelCallable`. However, it's not strictly necessary to use scipy. The callback must be represented with a tuple whose first element is a `PyCapsule`. The other elements are not used, but a reference to the tuple is held so this can be used to keep things alive). The capsule's pointer must be a function pointer to the code, the name must be the function signature, and a user-defined pointer may be set as the capsule's context.

For the place callback, the signature must be one of the following (exactly, with no whitespace changes):

- `"void (void *, size_t)"`
- `"void (void *, size_t, void *)"`

In the latter case, the capsule's context is provided as the final argument. The first two arguments are a pointer to *spead2::recv::chunk_place_data* and the size of that structure.

There are lots of ways to write compiled code and access the functions from Python: ctypes, cffi, cython, pybind11 are some of the options. One for which spead2 provided some extra support is numba:

`spead2.recv.numba.`**`chunk_place_data`**

Numba record type representing the C structure used in the chunk placement callback.

Numba doesn't (as of 0.54) support pointers in records, so the pointer fields are represented as integers. Use *spead2.numba.intp_to_voidptr()* to convert them to void pointers then `numba.carray()` to convert the void pointer to an array of the appropriate size and dtype.

`spead2.numba.`**`intp_to_voidptr`**(*typingctx*, *src*)

Convert an integer (of type intp or uintp) to a void pointer.

This is useful because numba doesn't (as of 0.54) support putting pointers into Records. They have to be smuggled in as intp, then converted to pointers with this function.

2.11.2 Reference

class `spead2.recv.`**`Chunk`**(***kwargs*)

The attributes can also be used as keywords arguments to the constructor. This class is designed to allow subclassing, and subclass properties will round-trip through the stream.

`data`

Data storage for a chunk. This can be set to any object that supports the Python buffer protocol, as long as it is contiguous and writable. Examples include (contiguous) numpy arrays, `bytearray` and `memoryview`. It can also be set to `None` to clear it.

`present`

Data storage for flags indicating presence of heaps within the chunk. This can be set to any object that supports the Python buffer protocol, as long as it is contiguous and writable. It can also be set to `None` to clear it.

`extra`

Data storage for extra data to be written by the place callback. This can be set to any object that supports the Python buffer protocol, as long as it is contiguous and writable. It can also be set to `None` to clear it.

`chunk_id`

The chunk ID determined by the placement function.

`stream_id`

Stream ID of the stream from which the chunk originated.

class `spead2.recv.`**`ChunkStreamConfig`**(***kwargs*)

Parameters for a `ChunkStream`. The configuration options can either be passed to the constructor (as keyword arguments) or set as properties after construction.

Parameters

- **items** (*List[int]*) – The items whose immediate values should be passed to the place function. Accessing this property returns a copy, so it cannot be updated with `append` or other mutating operations. Assign a complete list.
- **max_chunks** (*int*) – The maximum number of chunks that can be live at the same time. A value of 1 means that heaps must be received in order: once a chunk is started, no heaps from a previous chunk will be accepted.
- **place** (*tuple*) – See *Writing a place callback*.
- **max_heap_extra** (*int*) – The maximum amount of data a placement function may write to *spead2::recv::chunk_place_data::extra*.

Raises

ValueError – if *max_chunks* is zero.

enable_packet_presence(*payload_size: int*)

Enable the *packet presence feature*. The payload offset of each packet is divided by *payload_size* and added to the heap index before indexing *spead2.recv.Chunk.present*.

disable_packet_presence()

Disable the packet presence feature enabled by *enable_packet_presence()*.

packet_presence_payload_size

The *payload_size* if packet presence is enabled, or 0 if not.

DEFAULT_MAX_CHUNKS

Default value for `max_chunks`.

class spead2.recv.**ChunkRingbuffer**(*maxsize*)

Ringbuffer holding *Chunk*s. The interface is modelled on `Queue`, although the exceptions are different. It also implements the iterator protocol.

Once a chunk has been added to a ringbuffer it should not be accessed again until it is retrieved from a ringbuffer (either the same one, or more typically, a different one after it has been filled in by *ChunkRingStream*).

maxsize

Maximum capacity of the ringbuffer.

data_fd

A file descriptor that is readable when there is data available. This will not normally be used directly, but is used in the implementation of *spead2.recv.asyncio.ChunkRingbuffer*.

space_fd

A file descriptor that is readable when there is free space available. This will not normally be used directly, but is used in the implementation of *spead2.recv.asyncio.ChunkRingbuffer*.

qsize()

The current number of items in the ringbuffer.

empty()

True if the ringbuffer is empty, otherwise false.

full()

True if the ringbuffer is full, otherwise false.

get()

Retrieve an item from the ringbuffer, blocking if necessary.

Raises
spead2.Stopped – if the ringbuffer was stopped before an item became available.

get_nowait()

Retrieve an item from the ringbuffer, raising an exception if none is available.

Raises

- **spead2.Stopped** – if the ringbuffer is stopped and empty.
- **spead2.Empty** – if the ringbuffer is empty.

put(*chunk*)

Put an item into the ringbuffer, blocking until there is space if necessary.

Raises
spead2.Stopped – if the ringbuffer was stopped before space became available.

put_nowait(*chunk*)

Put an item into the ringbuffer, raising an exception if there is no space available.

Raises

- **spead2.Stopped** – if the ringbuffer is stopped.
- **spead2.Full** – if the ringbuffer is full.

stop()

Shut down the ringbuffer. Producers will no longer be able to add new items. Consumers will be able to retrieve existing items, after which they will receive `spead2.Stopped`, and iterators will terminate.

Returns true if this call stopped the ringbuffer, otherwise false.

add_producer()

Register a new producer. Producers only need to call this if they want to call *remove_producer()*.

remove_producer()

Indicate that a producer registered with *add_producer()* is finished with the ringbuffer. If this was the last producer, the ringbuffer is stopped. Returns true if this call stopped the ringbuffer, otherwise false.

class spead2.recv.asyncio.**ChunkRingbuffer**(*maxsize*)

Asynchronous chunk ringbuffer.

It provides asynchronous versions of the blocking functions, and the asychronous iterator protocol.

async get()

Get a chunk from the ringbuffer asynchronously.

Raises
spead2.Stopped – if the ringbuffer is stopped before a chunk becomes available

async put(*chunk*)

Put an item into the ringbuffer asynchronously.

Raises

spead2.Stopped – if the ringbuffer is stopped

class spead2.recv.**ChunkRingStream**(*thread_pool*, *config*, *chunk_config*, *data_ringbuffer*, *free_ringbuffer*)

Stream that works on chunks. While it is not a direct subclass, it implements most of the same functions as *spead2.recv.Stream*, in particular for adding transports.

Parameters

- **thread_pool** (*spead2.ThreadPool*) – Thread pool handling the I/O
- **config** (*spead2.recv.StreamConfig*) – Stream configuration
- **chunk_config** (*spead2.recv.ChunkStreamConfig*) – Chunking configuration
- **data_ringbuffer** (*spead2.recv.ChunkRingbuffer*) – Ringbuffer onto which the stream will place completed chunks.
- **free_ringbuffer** (*spead2.recv.ChunkRingbuffer*) – Ringbuffer from which the stream will obtain new chunks.

data_ringbuffer

The data ringbuffer given to the constructor.

free_ringbuffer

The free ringbuffer given to the constructor.

add_free_chunk(*chunk*)

Add a chunk to the free ringbuffer. This takes care of zeroing out the *Chunk.present* array, and it will suppress the spead2.Stopped exception if the free ringbuffer has been stopped.

If the free ring is full, it will raise spead2.Full rather than blocking. The free ringbuffer should be constructed with enough slots that this does not happen.

2.12 Chunking stream groups

For an overview, refer to *Chunking stream groups*. This page is a reference for the Python API. It extends the API for *chunks*.

class spead2.recv.**ChunkStreamGroupConfig**(***kwargs*)

Parameters for a chunk stream group. The configuration options can either be passed to the constructor (as keyword arguments) or set as properties after construction.

Parameters

- **max_chunks** (*int*) – The maximum number of chunks that can be live at the same time.
- **eviction_mode** (EvictionMode) – The chunk eviction mode.

class EvictionMode

Eviction mode when it is necessary to advance the group window. See the *overview* for more details.

LOSSY

force streams to release incomplete chunks

LOSSLESS

a chunk will only be marked ready when all streams have marked it ready

class spead2.recv.**ChunkStreamRingGroup**(*config*, *data_ringbuffer*, *free_ringbuffer*)

Stream group that uses ringbuffers to manage chunks.

When a fresh chunk is needed, it is retrieved from a ringbuffer of free chunks (the "free ring"). When a chunk is flushed, it is pushed to a "data ring". These may be shared between groups, but both will be stopped as soon as any of the members streams are stopped. The intended use case is parallel groups that are started and stopped together.

It behaves like a `Sequence` of the contained streams.

Parameters

- **config** (*spead2.recv.ChunkStreamGroupConfig*) – Group configuration
- **data_ringbuffer** (*spead2.recv.ChunkRingbuffer*) – Ringbuffer onto which the completed chunks are placed.
- **free_ringbuffer** (*spead2.recv.ChunkRingbuffer*) – Ringbuffer from which new chunks are obtained.

data_ringbuffer

The data ringbuffer given to the constructor.

free_ringbuffer

The free ringbuffer given to the constructor.

add_free_chunk(*chunk*)

Add a chunk to the free ringbuffer. This takes care of zeroing out the *Chunk.present* array, and it will suppress the `spead2.Stopped` exception if the free ringbuffer has been stopped.

If the free ring is full, it will raise `spead2.Full` rather than blocking. The free ringbuffer should be constructed with enough slots that this does not happen.

emplace_back(*thread_pool*, *config*, *chunk_stream_config*)

Add a new stream.

Parameters

- **thread_pool** (*spead2.ThreadPool*) – Thread pool handling the I/O
- **config** (*spead2.recv.StreamConfig*) – Stream configuration
- **chunk_config** (*spead2.recv.ChunkStreamConfig*) – Chunking configuration

Return type

spead2.recv.ChunkStreamGroupMember

class spead2.recv.**ChunkStreamGroupMember**

A component stream in a *ChunkStreamRingGroup*. This class cannot be instantiated directly. Use *ChunkStreamRingGroup.emplace_back()* instead.

It provides the same methods for adding readers as *spead2.recv.Stream*.

CHAPTER

THREE

C++ API FOR SPEAD2

The C++ API is at a lower level than the Python API. In particular, item values are treated as uninterpreted binary blobs. The protocol is directly tied to numpy's type system, so it is not practical to implement this in C++. The C++ API is thus best suited to situations which require the maximum possible performance and where the data formats can be fixed in advance.

There is also no equivalent to the *spead2.ItemGroup* and *spead2.send.HeapGenerator* classes. The user is responsible for maintaining previously seen descriptors (if they are desired) and tracking which descriptors and items need to be inserted into heaps.

The C++ documentation is far from complete. As a first step, consult the Python documentation; in many cases it is just wrapping the C++ interface with Pythonic names, whereas the C++ interface uses lowercase with underscores for all names. If that doesn't help, consult the Doxygen-style comments in the source code.

The compiler and link flags necessary for compiling and linking against spead2 can be found with **pkg-config** i.e.,

- `pkg-config --cflags spead2` to get the compiler flags
- `pkg-config --libs --static spead2` to get the linker flags

Note that when installed with the default setup on a GNU/Linux system, the `spead2.pc` file is installed outside **pkg-config**'s default search path, and you need to set `PKG_CONFIG_PATH` to `/usr/local/lib/pkgconfig` first.

3.1 C++ API stability

The C++ API is less stable between versions than the Python API. The most-derived classes defining specific transports are expected to be stable. Applications that subclass the base classes to define new transports may be broken by future API changes, as there is still room for improvement in the API between these classes and the core.

3.2 Asynchronous I/O

The C++ API uses Boost.Asio for asynchronous operations. There is a *spead2::thread_pool* class (essentially the same as the Python *spead2.ThreadPool* class). However, it is not required to use this, and you may for example run everything in one thread to avoid multi-threading issues.

class **thread_pool**

Combination of a `boost::asio::io_service` with a set of threads to handle the callbacks.

The threads are created by the constructor and shut down and joined in the destructor.

Subclassed by spead2::thread_pool_wrapper

Public Functions

thread_pool(int num_threads, const std::vector<int> &affinity)

Construct with explicit core affinity for the threads.

The *affinity* list can be shorter or longer than *num_threads*. Threads are allocated in round-robin fashion to cores. Failures to set affinity are logged but do not cause an exception.

inline boost::asio::io_service &**get_io_service**()

Retrieve the embedded io_service.

void **stop**()

Shut down the thread pool.

Public Static Functions

static void **set_affinity**(int core)

Set CPU affinity of current thread.

Classes that perform asynchronous operations take a parameter of type `spead2::io_service_ref`. This can be (implicitly) initialised from either a `boost::asio::io_service` reference, a `spead2::thread_pool` reference, or a `std::shared_ptr<spead2::thread_pool>`. In the last case, the receiving class retains a copy of the shared pointer, providing convenient lifetime management of a thread pool.

class **io_service_ref**

A helper class that holds a reference to a `boost::asio::io_service`, and optionally a shared pointer to a *thread_pool*.

It is normally not explicitly constructed, but other classes that need an `io_service` take it as an argument and store it so that they can accept any of `io_service`, `thread_pool` or `std::shared_ptr<thread_pool>`, and in the last case they hold on to the reference.

Public Functions

io_service_ref(boost::asio::io_service&)

Construct from a reference to an `io_service`.

io_service_ref(*thread_pool*&)

Construct from a reference to a *thread_pool*.

template<typename **T**, typename **SFINAE** = std::enable_if_t<std::is_convertible_v<*T**, *thread_pool**>>>
io_service_ref(std::shared_ptr<*T*>)

Construct from a shared pointer to a *thread_pool*.

This is templated so that it will also accept a shared pointer to a subclass of *thread_pool*.

boost::asio::io_service &**operator***() const

Return the referenced `io_service`.

boost::asio::io_service ***operator->**() const

Return a pointer to the referenced `io_service`.

std::shared_ptr<*thread_pool*> **get_shared_thread_pool**() const &

Return the shared pointer to the *thread_pool*, if constructed from one.

std::shared_ptr<*thread_pool*> &&**get_shared_thread_pool**() &&

Return the shared pointer to the *thread_pool*, if constructed from one.

This overload returns an rvalue reference, allowing the shared pointer to be moved out of a temporary.

A number of the APIs use callbacks. These follow the usual Boost.Asio guarantee that they will always be called from threads running `boost::asio::io_service::run()`. If using a `thread_pool`, this will be one of the threads managed by the pool. Additionally, callbacks for a specific stream are serialised, but there may be concurrent callbacks associated with different streams.

3.3 Receiving

3.3.1 Heaps

Unlike the Python bindings, the C++ bindings expose three heap types: *live heaps* (`spead2::recv::live_heap`) are used for heaps being constructed, and may be missing data; *frozen heaps* (`spead2::recv::heap`) always have all their data; and *incomplete heaps* (`spead2::recv::incomplete_heap`) are frozen heaps that are missing data. Frozen heaps can be move-constructed from live heaps, which will typically be done in the callback.

class **live_heap**

A SPEAD heap that is in the process of being received.

Once it is fully received, it is converted to a *heap* for further processing.

Any SPEAD-64-* flavour can be used, but all packets in the heap must use the same flavour. It may be possible to relax this, but it hasn't been examined, and may cause issues for decoding descriptors (whose format depends on the flavour).

A heap can be:

- complete: a heap length item was found in a packet, and we have received all the payload corresponding to it. No more packets are expected.
- contiguous: the payload we have received is a contiguous range from 0 up to some amount, and cover all items described in the item pointers.
- incomplete: not contiguous A complete heap is also contiguous, but not necessarily the other way around. Only contiguous heaps can be frozen to *heap*, and only incomplete heaps can be frozen to *incomplete_heap*.

Public Functions

bool **is_complete**() const

True if the heap is complete.

bool **is_contiguous**() const

True if the heap is contiguous.

bool **is_end_of_stream**() const

True if an end-of-stream heap control item was found.

inline s_item_pointer_t **get_cnt**() const

Retrieve the heap ID.

inline bug_compat_mask **get_bug_compat**() const

Get protocol bug compatibility flags.

class **heap** : public spead2::recv::heap_base

Received heap that has been finalised.

Public Functions

explicit **heap** (*live_heap* &&h)

Freeze a heap, which must satisfy *live_heap::is_contiguous*.

The original heap is destroyed.

descriptor **to_descriptor** () const

Extract descriptor fields from the heap.

Any missing fields are default-initialized. This should be used on a heap constructed from the content of a descriptor item.

The original PySPEAD package (version 0.5.2) does not follow the specification here. The macros in common_defines.h can be used to control whether to interpret the specification or be bug-compatible.

The protocol allows descriptors to use immediate-mode items, but the decoding of these into variable-length strings is undefined. This implementation will discard such descriptor fields.

std::vector<*descriptor*> **get_descriptors** () const

Extract and decode descriptors from this heap.

inline s_item_pointer_t **get_cnt** () const

Get heap ID.

inline const flavour &**get_flavour** () const

Get protocol flavour used.

inline const std::vector<*item*> &**get_items** () const

Get the items from the heap.

This includes descriptors, but excludes any items with ID <= 4.

bool **is_ctrl_item** (ctrl_mode value) const

Convenience function to check whether any of the items is a `STREAM_CTRL_ID` item with value *value*.

bool **is_start_of_stream** () const

Convenience function to check whether any of the items is a `CTRL_STREAM_START`.

bool **is_end_of_stream** () const

Convenience function to check whether any of the items is a `CTRL_STREAM_STOP`.

inline const *memory_allocator*::pointer &**get_payload** () const

Get the payload pointer.

This will return an empty pointer unless *keep_payload* was set in the constructor. This is not normally needed, but has applications in some advanced use cases.

class **incomplete_heap** : public spead2::recv::heap_base

Received heap that has been finalised, but which is missing data.

The payload and any items that refer to the payload are discarded.

Public Functions

incomplete_heap(*live_heap* &&h, bool keep_payload, bool keep_payload_ranges)

Freeze a heap.

The original heap is destroyed.

Parameters

- **h** – The heap to freeze.
- **keep_payload** – If true, transfer the payload memory allocation from the live heap to this object. If false, discard it.
- **keep_payload_ranges** – If true, store information that allows *get_payload_ranges* to work.

inline s_item_pointer_t **get_heap_length**() const

Heap payload length encoded in packets (-1 for unknown)

inline s_item_pointer_t **get_received_length**() const

Number of bytes of payload received.

std::vector<std::pair<s_item_pointer_t, s_item_pointer_t>> **get_payload_ranges**() const

Return a list of contiguous ranges of payload that were received.

This is intended for special cases where a custom memory allocator was used to channel the payload into a caller-managed area, so that the caller knows which parts of that area have been filled in.

If *keep_payload_ranges* was `false` in the constructor, returns an empty list.

inline s_item_pointer_t **get_cnt**() const

Get heap ID.

inline const flavour &**get_flavour**() const

Get protocol flavour used.

inline const std::vector<*item*> &**get_items**() const

Get the items from the heap.

This includes descriptors, but excludes any items with ID <= 4.

bool **is_ctrl_item**(ctrl_mode value) const

Convenience function to check whether any of the items is a `STREAM_CTRL_ID` item with value *value*.

bool **is_start_of_stream**() const

Convenience function to check whether any of the items is a `CTRL_STREAM_START`.

bool **is_end_of_stream**() const

Convenience function to check whether any of the items is a `CTRL_STREAM_STOP`.

inline const *memory_allocator*::pointer &**get_payload**() const

Get the payload pointer.

This will return an empty pointer unless *keep_payload* was set in the constructor. This is not normally needed, but has applications in some advanced use cases.

struct **item**

An item extracted from a heap.

Subclassed by spead2::recv::item_wrapper

Public Members

s_item_pointer_t **id**

Item ID.

std::uint8_t ***ptr**

Start of memory containing value.

std::size_t **length**

Length of memory.

item_pointer_t **immediate_value**

The immediate interpreted as an integer (undefined if not immediate)

bool **is_immediate**

Whether the item is immediate.

struct **descriptor**

An unpacked descriptor.

If *numpy_header* is non-empty, it overrides *format* and *shape*.

Public Members

s_item_pointer_t **id** = 0

SPEAD ID.

std::string **name**

Short name.

std::string **description**

Long description.

std::vector<std::pair<char, s_item_pointer_t>> **format**

Legacy format.

Each element is a specifier character (e.g. ‘u’ for unsigned) and a bit width.

std::vector<s_item_pointer_t> **shape**

Shape.

Elements are either non-negative, or -1 is used to indicate a variable-length size. At most one dimension may be variable-length.

std::string **numpy_header**

Description in the format used in .npy files.

3.3.2 Streams

At the lowest level, heaps are given to the application via a callback to a virtual function. While this callback is running, no new packets can be received from the network socket, so this function needs to complete quickly to avoid data loss when using UDP. To use this interface, subclass `spead2::recv::stream` and implement `heap_ready()` and optionally override `stop_received()`.

Note that some public functions are incorrectly listed as protected below due to limitations of the documentation tools.

class **stream_config**

Parameters for a receive stream.

Public Functions

stream_config &**set_max_heaps**(std::size_t max_heaps)

Set maximum number of partial heaps that can be live at one time (per substream).

This affects how intermingled heaps can be (due to out-of-order packet delivery) before heaps get dropped.

inline std::size_t **get_max_heaps**() const

Get maximum number of partial heaps that can be live at one time.

stream_config &**set_substreams**(std::size_t substreams)

Set number of substreams.

The substream is determined by taking the heap cnt modulo the number of substreams. The value set by *set_max_heaps* applies independently for each substream.

inline std::size_t **get_substreams**() const

Get number of substreams.

stream_config &**set_memory_allocator**(std::shared_ptr<*memory_allocator*> allocator)

Set an allocator to use for allocating heap memory.

inline const std::shared_ptr<*memory_allocator*> &**get_memory_allocator**() const

Get allocator for allocating heap memory.

stream_config &**set_memcpy**(*packet_memcpy_function* memcpy)

Set an alternative memcpy function for copying heap payload.

stream_config &**set_memcpy**(memcpy_function memcpy)

Set an alternative memcpy function for copying heap payload.

stream_config &**set_memcpy**(memcpy_function_id id)

Set builtin memcpy function to use for copying heap payload.

inline const *packet_memcpy_function* &**get_memcpy**() const

Get memcpy function for copying heap payload.

stream_config &**set_stop_on_stop_item**(bool stop)

Set whether to stop the stream when a stop item is received.

inline bool **get_stop_on_stop_item**() const

Get whether to stop the stream when a stop item is received.

stream_config &**set_allow_unsized_heaps**(bool allow)

Set whether to allow heaps without HEAP_LENGTH.

inline bool **get_allow_unsized_heaps**() const

Get whether to allow heaps without HEAP_LENGTH.

stream_config &**set_allow_out_of_order**(bool allow)

Set whether to allow out-of-order packets within a heap.

inline bool **get_allow_out_of_order**() const

Get whether to allow out-of-order packets within a heap.

stream_config &**set_bug_compat**(bug_compat_mask bug_compat)

Set bug compatibility flags.

inline bug_compat_mask **get_bug_compat**() const

Get bug compatibility flags.

stream_config &**set_stream_id**(std::uintptr_t stream_id)

Set a stream ID.

inline std::uintptr_t **get_stream_id**() const

Get the stream ID.

stream_config &**set_explicit_start**(bool explicit_start)

Set the explicit start control.

If explicit start is enabled, no data will be received on the stream until *stream::start* is called, and no readers can be added afterwards.

inline bool **get_explicit_start**() const

Get the explicit start flag.

std::size_t **add_stat**(std::string name, *stream_stat_config*::*mode* mode = *stream_stat_config*::*mode*::*COUNTER*)

Add a new custom statistic.

Returns the index to use with *stream_stats*.

Throws

`std::invalid_argument` – if *name* already exists.

inline const std::vector<*stream_stat_config*> &**get_stats**() const

Get the stream statistics (including the core ones)

std::size_t **get_stat_index**(const std::string &name) const

Helper to get the index of a specific statistic.

Throws

`std::out_of_range` – if *name* is not a known statistic.

inline std::size_t **next_stat_index**() const

The index that will be returned by the next call to *add_stat*.

class **stream** : protected spead2::recv::stream_base

Stream that is fed by subclasses of reader.

The public interface to this class is thread-safe.

Subclassed by *spead2::recv::chunk_stream*, *spead2::recv::chunk_stream_group_member*, spead2::recv::ring_stream_base

Public Functions

template<typename **T**, typename ...**Args**>
inline void **emplace_reader**(*Args*&&... args)

Add a new reader by passing its constructor arguments, excluding the initial *io_service* and *owner* arguments.

void **start**()

Start the stream.

This is only needed if the config specifies explicit start (see *stream_config::set_explicit_start*). In that case, no new readers can be added after starting the stream.

virtual void **stop**()

Stop the stream.

After this returns, the io_service may still have outstanding completion handlers, but they should be no-ops when they're called.

In most cases subclasses should override stop_received rather than this function. However, if heap_ready can block indefinitely, this function should be overridden to unblock it before calling the base implementation.

inline const *stream_config* &**get_config**() const

Get the stream's configuration.

stream_stats **get_stats**() const

Return statistics about the stream.

See the Python documentation.

Protected Functions

inline const *stream_config* &**get_config**() const

Get the stream's configuration.

void **flush**()

Flush the collection of live heaps, passing them to heap_ready.

stream_stats **get_stats**() const

Return statistics about the stream.

See the Python documentation.

A potentially more convenient interface is `spead2::recv::ring_stream<Ringbuffer>`, which places received heaps into a fixed-size thread-safe ring buffer. Another thread can then pull from this ring buffer in a loop. The template parameter selects the ringbuffer implementation. The default is a good light-weight choice, but if you need to use `select()`-like functions to wait for data, you can use `spead2::ringbuffer<spead2::recv::live_heap, spead2::semaphore_fd, spead2::semaphore>`.

class **ring_stream_config**

Parameters for configuring *ring_stream*.

Subclassed by spead2::recv::ring_stream_config_wrapper

Public Functions

ring_stream_config &**set_heaps**(std::size_t heaps)

Set capacity of the ring buffer.

inline std::size_t **get_heaps**() const

Get capacity of the ring buffer.

ring_stream_config &**set_contiguous_only**(bool contiguous_only)

Set whether only contiguous heaps are pushed to the ring buffer.

inline bool **get_contiguous_only**() const

Get whether only contiguous heaps are pushed to the ring buffer.

template<typename **Ringbuffer** = *ringbuffer*<*live_heap*>>

class **ring_stream** : public spead2::recv::ring_stream_base

Specialisation of *stream* that pushes its results into a ringbuffer.

The ringbuffer class may be replaced, but must provide the same interface as *ringbuffer*. If the ring buffer fills up, add_packet will block the reader.

On the consumer side, heaps are automatically frozen as they are extracted.

This class is thread-safe.

Public Functions

explicit **ring_stream**(*io_service_ref* io_service, const *stream_config* &config = *stream_config*(), const *ring_stream_config* &ring_config = *ring_stream_config*())

Constructor.

Parameters

- **io_service** – I/O service (also used by the readers).
- **config** – Stream configuration
- **ring_config** – Ringbuffer configuration

heap **pop**()

Wait until a contiguous heap is available, freeze it, and return it; or until the stream is stopped.

Throws

ringbuffer_stopped – if stop has been called and there are no more contiguous heaps.

template<typename ...**SemArgs**>
live_heap **pop_live**(*SemArgs*&&... sem_args)

Wait until a heap is available and return it; or until the stream is stopped.

Parameters

sem_args – Arbitrary arguments to pass to the data semaphore

Throws

ringbuffer_stopped – if stop has been called and there are no more heaps.

heap **try_pop**()

Like *pop*, but if no contiguous heap is available, throws *spead2::ringbuffer_empty*.

Throws

- `ringbuffer_empty` – if there is no contiguous heap available, but the stream has not been stopped
- `ringbuffer_stopped` – if stop has been called and there are no more contiguous heaps.

live_heap **try_pop_live**()

Like *pop_live*, but if no heap is available, throws *spead2::ringbuffer_empty*.

Throws

- `ringbuffer_empty` – if there is no heap available, but the stream has not been stopped
- `ringbuffer_stopped` – if stop has been called and there are no more heaps.

spead2::detail::ringbuffer_iterator<*ring_stream*> **begin**()

Start iteration over the heaps in the ringbuffer.

This does not return a full-fledged iterator. It is intended only to enable a range-based for loop over the stream, and any other use of the iterator is unsupported.

For example: `for (spead2::recv::heap heap : stream) { ... }`

spead2::detail::ringbuffer_sentinel **end**()

End iterator (see *begin*).

inline const *ring_stream_config* &**get_ring_config**() const

Get the ringbuffer configuration passed to the constructor.

3.3.3 Readers

Reader classes are constructed inside a stream by calling `spead2::recv::stream::emplace_reader()`.

class **udp_reader** : public spead2::recv::udp_reader_base

Asynchronous stream reader that receives packets over UDP.

Public Functions

udp_reader(*stream* &owner, const boost::asio::ip::udp::endpoint &endpoint, std::size_t max_size = default_max_size, std::size_t buffer_size = default_buffer_size)

Constructor.

If *endpoint* is a multicast address, then this constructor will subscribe to the multicast group, and also set `SO_REUSEADDR` so that multiple sockets can be subscribed to the multicast group.

Parameters

- **owner** – Owning stream
- **endpoint** – Address on which to listen
- **max_size** – Maximum packet size that will be accepted.
- **buffer_size** – Requested socket buffer size. Note that the operating system might not allow a buffer size as big as the default.

udp_reader (*stream* &owner, const boost::asio::ip::udp::endpoint &endpoint, std::size_t max_size, std::size_t buffer_size, const boost::asio::ip::address &interface_address)

Constructor with explicit interface address (IPv4 only).

This overload is designed for use with multicast, but can also be used with a unicast endpoint as long as the address matches the interface address.

When a multicast group is used, the socket will have `SO_REUSEADDR` set, so that multiple sockets can all listen to the same multicast stream. If you want to let the system pick the interface for the multicast subscription, use `boost::asio::ip::address_v4::any()`, or use the default constructor.

Parameters

- **owner** – Owning stream
- **endpoint** – Address and port
- **max_size** – Maximum packet size that will be accepted.
- **buffer_size** – Requested socket buffer size.
- **interface_address** – Address of the interface which should join the group

Throws

- `std::invalid_argument` – If *endpoint* is not an IPv4 multicast address and does not match *interface_address*.
- `std::invalid_argument` – If *interface_address* is not an IPv4 address

udp_reader (*stream* &owner, const boost::asio::ip::udp::endpoint &endpoint, std::size_t max_size, std::size_t buffer_size, unsigned int interface_index)

Constructor with explicit multicast interface index (IPv6 only).

The socket will have `SO_REUSEADDR` set, so that multiple sockets can all listen to the same multicast stream. If you want to let the system pick the interface for the multicast subscription, set *interface_index* to 0, or use the standard constructor.

See also:

if_nametoindex(3)

Parameters

- **owner** – Owning stream
- **endpoint** – Multicast group and port
- **max_size** – Maximum packet size that will be accepted.
- **buffer_size** – Requested socket buffer size.
- **interface_index** – Address of the interface which should join the group

udp_reader (*stream* &owner, boost::asio::ip::udp::socket &&socket, std::size_t max_size = default_max_size)

Constructor using an existing socket.

This allows socket options (e.g., multicast subscriptions) to be fine-tuned by the caller. The socket must already be bound to the desired endpoint. There is no special handling of multicast subscriptions or socket buffer sizes here.

Parameters

- **owner** – Owning stream
- **socket** – Existing socket which will be taken over. It must use the same I/O service as *owner*.
- **max_size** – Maximum packet size that will be accepted.

class **tcp_reader** : public spead2::recv::reader

Asynchronous stream reader that receives packets over TCP.

Public Functions

tcp_reader (*stream* &owner, const boost::asio::ip::tcp::endpoint &endpoint, std::size_t max_size = default_max_size, std::size_t buffer_size = default_buffer_size)

Constructor.

Parameters

- **owner** – Owning stream
- **endpoint** – Address on which to listen
- **max_size** – Maximum packet size that will be accepted.
- **buffer_size** – Requested socket buffer size. Note that the operating system might not allow a buffer size as big as the default.

tcp_reader (*stream* &owner, boost::asio::ip::tcp::acceptor &&acceptor, std::size_t max_size = default_max_size)

Constructor using an existing acceptor object.

This allows acceptor objects to be created and fine-tuned by users before handing them over. The acceptor object must be already bound.

Parameters

- **owner** – Owning stream
- **acceptor** – Acceptor object, must be bound
- **max_size** – Maximum packet size that will be accepted.

Private Functions

tcp_reader (*stream* &owner, boost::asio::ip::tcp::acceptor &&acceptor, std::size_t max_size, std::size_t buffer_size)

Base constructor, used by the other constructors.

Parameters

- **owner** – Owning stream
- **acceptor** – Acceptor object, must be bound
- **max_size** – Maximum packet size that will be accepted.
- **buffer_size** – Requested socket buffer size. Note that the operating system might not allow a buffer size as big as the default.

class **mem_reader** : public spead2::recv::reader

Reader class that feeds data from a memory buffer to a stream.

The caller must ensure that the underlying memory buffer is not destroyed before this class.

Note: For simple cases, use mem_to_stream instead. This class is only necessary if one wants to plug in to a *stream*.

Subclassed by spead2::recv::buffer_reader

class **udp_pcap_file_reader** : public spead2::recv::udp_reader_base

Reader class that feeds data from a pcap file to a stream.

An optional filter selects a subset of the packets in the capture file.

Public Functions

udp_pcap_file_reader (*stream* &owner, const std::string &filename, const std::string &filter = "")

Constructor.

Parameters

- **owner** – Owning stream
- **filename** – Filename of the capture file
- **filter** – Filter to apply to packets from the capture file

Throws

`std::runtime_error` – if *filename* could not read

3.3.4 Memory allocators

In addition to the memory allocators described in *Memory allocators*, new allocators can be created by subclassing `spead2::memory_allocator`. For an allocator set on a stream, a pointer to a `spead2::recv::packet_header` is passed as a hint to the allocator, allowing memory to be placed according to information in the packet. Note that if the `stream_config` has been configured to allow out-of-order packets then this could be any packet from the heap, rather than the first one.

class **memory_allocator** : public std::enable_shared_from_this<*memory_allocator*>

Polymorphic class for managing memory allocations in a memory pool.

This can be overloaded to provide custom memory allocations.

Subclassed by spead2::memory_pool, spead2::mmap_allocator, spead2::recv::detail::chunk_stream_allocator< CM >, spead2::unittest::legacy_allocator, spead2::unittest::mock_allocator

Public Functions

virtual pointer **allocate**(std::size_t size, void *hint)

Allocate *size* bytes of memory.

The default implementation uses `new` and pre-faults the memory.

The `pointer` type uses *memory_allocator::deleter* as the deleter. If the memory needs to be freed by something other than `delete[]` then pass a function object when constructing the pointer.

Parameters

- **size** – Number of bytes to allocate
- **hint** – Usage-dependent extra information

Throws

`std::bad_alloc` – if allocation failed

Returns

Pointer to newly allocated memory

Private Functions

virtual void **free**(std::uint8_t *ptr, void *user)

Free memory previously returned from *allocate*.

This is kept only for backwards compatibility, so that existing allocators that override it will continue to work.

Parameters

- **ptr** – Value returned by *allocate*
- **user** – User-defined handle stored in the deleter by *allocate*

class **deleter** : public std::function<void(std::uint8_t*)>

Deleter for pointers allocated by this allocator.

This class derives from `std::function`, so it can be constructed with any deleter at run time. In particular, this means that `std::unique_ptr<std::uint8_t[], D>` can be converted to `memory_allocator::pointer` for any deleter `D`.

For backwards compatibility, it can also be constructed from a shared pointer to an allocator and an arbitrary void pointer, in which case the deletion is performed by calling *memory_allocator::free*.

Public Functions

const std::shared_ptr<*memory_allocator*> &**get_allocator**() const

If the backwards-compatibility constructor was used, return the stored allocator.

Otherwise, returns a null pointer.

void ***get_user**() const

If the backwards-compatibility constructor was used, return the stored user pointer.

Otherwise, returns a null pointer.

The file `examples/gdrapi_example.cu` in the spead2 source distribution shows an example of using a custom memory allocator to allocate memory for heaps on the GPU.

3.3.5 Custom memory scatter

In specialised high-bandwidth cases, the overhead of assembling heaps in temporary storage before scattering the data into other arrangements can be very high. It is possible (since 1.11) to take complete control over the transfer of the payload of the SPEAD packets. Before embarking on such an approach, be sure you have a good understanding of the SPEAD protocol, particularly packets, heaps, item pointers and payload.

In the simplest case, each heap needs to be written to some special or pre-allocated storage, but in a contiguous fashion. In this case it is sufficient to provide a custom allocator (see above), which will return a pointer to the target storage.

In more complex cases, the contents of each heap, or even each packet, needs to be scattered to discontiguous storage areas. In this case, one can additionally override the memory copy function with `set_memcpy()` and providing a *packet_memcpy_function*.

typedef std::function<void(const spead2::*memory_allocator*::pointer &allocation, const packet_header &packet)>
`spead2::recv::`**`packet_memcpy_function`**

It takes a pointer to the start of the heap's allocation (as returned by the allocator) and the packet metadata. The default implementation is equivalent to the following:

```
void copy(const spead2::memory_allocator::pointer &allocation, const packet_header &
↪packet)
{
    memcpy(allocation.get() + packet.payload_offset, packet.payload, packet.payload_
↪length);
}
```

Note that when providing your own memory copy and allocator, you don't necessarily need the allocator to actually return a pointer to payload memory. It could, for example, populate a structure that guides the copy, and return a pointer to that; or it could return a null pointer. There are some caveats though:

1. If the sender doesn't provide the heap length item, then spead2 may need to make multiple allocations of increasing size as the heap grows, and each time it will copy (with standard memcpy, rather than your custom one) the old content to the new. Assuming you aren't expecting such packets, you can reject them using `set_allow_unsized_heaps()`.
2. `spead2::recv::heap_base::get_items()` constructs pointers to the items on the assumption of the default memcpy function, so if your replacement doesn't copy things to the same place, you obviously won't be able to use those pointers. Note that *`get_descriptors()`* will also not be usable.

See `examples/gdrapi_example.cu` in the spead2 source distribution for an example that copies data to a GPU.

3.3.6 Statistics

See *Receiver stream statistics* for an overview of statistics.

class **`stream_stats`**

Statistics about a stream.

Both a vector-like interface (indexing and *size*) and a map-like interface (with iterators and *find*) are provided. The iterators support random traversal, but are not technically random access iterators because dereferencing does not return a reference.

The public members provide direct access to the core statistics, for backwards compatibility. New code is advised to use the other interfaces. Indices for core statistics are available in *stream_stat_indices*.

Public Types

using **key_type** = std::string

using **mapped_type** = std::uint64_t

using **value_type** = std::pair<const std::string, std::uint64_t>

using **size_type** = std::size_t

using **difference_type** = std::ptrdiff_t

using **reference** = *value_type*&

using **const_reference** = const *value_type*&

using **pointer** = *value_type**

using **const_pointer** = const *value_type**

using **iterator** = detail::stream_stats_iterator<*stream_stats*, std::pair<const std::string&, std::uint64_t&>>

using **const_iterator** = detail::stream_stats_iterator<const *stream_stats*, const std::pair<const std::string&, const std::uint64_t&>>

using **reverse_iterator** = std::reverse_iterator<*iterator*>

using **const_reverse_iterator** = std::reverse_iterator<*const_iterator*>

Public Functions

stream_stats()

Construct with the default set of statistics, and all zero values.

explicit **stream_stats**(std::shared_ptr<const std::vector<*stream_stat_config*>> config)

Construct with all zero values.

stream_stats(std::shared_ptr<const std::vector<*stream_stat_config*>> config, std::vector<std::uint64_t> values)

Construct with provided values.

stream_stats(const *stream_stats* &other)

stream_stats &**operator=**(const *stream_stats* &other)

stream_stats(*stream_stats* &&other) = default

inline const std::vector<*stream_stat_config*> &**get_config**() const

Get the configuration of the statistics.

inline bool **empty**() const

Whether the container is empty.

inline std::size_t **size**() const

Get the number of statistics.

inline std::uint64_t &**operator[]**(std::size_t index)

Access a statistic by index.

If index is out of range, behaviour is undefined.

inline const std::uint64_t &**operator[]**(std::size_t index) const

Access a statistic by index.

If index is out of range, behaviour is undefined.

inline std::uint64_t &**at**(std::size_t index)

Access a statistic by index.

Throws
`std::out_of_range` – if index is out of range.

inline const std::uint64_t &**at**(std::size_t index) const

Access a statistic by index.

Throws
`std::out_of_range` – if index is out of range.

std::uint64_t &**operator[]**(const std::string &name)

Access a statistic by name.

Throws
`std::out_of_range` – if *name* is not the name of a statistic

const std::uint64_t &**operator[]**(const std::string &name) const

Access a statistic by name.

Throws
`std::out_of_range` – if *name* is not the name of a statistic

std::uint64_t &**at**(const std::string &name)

Access a statistic by name.

Throws
`std::out_of_range` – if *name* is not the name of a statistic

const std::uint64_t &**at**(const std::string &name) const

Access a statistic by name.

Throws
`std::out_of_range` – if *name* is not the name of a statistic

inline *const_iterator* **cbegin**() const noexcept

inline *const_iterator* **cend**() const noexcept

inline *iterator* **begin**() noexcept

inline *iterator* **end**() noexcept

inline *const_iterator* **begin**() const noexcept

inline *const_iterator* **end**() const noexcept

inline *const_reverse_iterator* **crbegin**() const noexcept

inline *const_reverse_iterator* **crend**() const noexcept

inline *reverse_iterator* **rbegin**() noexcept

inline *reverse_iterator* **rend**() noexcept

inline *const_reverse_iterator* **rbegin**() const noexcept

inline *const_reverse_iterator* **rend**() const noexcept

iterator **find**(const std::string &name)

Find element with the given name.

If not found, returns `end()`.

const_iterator **find**(const std::string &name) const

Find element with the given name.

If not found, returns `end()`.

std::size_t **count**(const std::string &name) const

Return the number of elements matching *name* (0 or 1).

stream_stats **operator+**(const *stream_stats* &other) const

Combine two sets of statistics.

Each statistic is combined according to its mode.

See also:

stream_stat_config::mode

Throws

`std::invalid_argument` – if *other* has a different list of statistics

stream_stats &**operator+=**(const *stream_stats* &other)

Combine another set of statistics with this one.

Each statistic is combined according to its mode.

See also:

stream_stat_config::mode

Throws

`std::invalid_argument` – if *other* has a different list of statistics

Public Members

std::uint64_t &**heaps**

std::uint64_t &**incomplete_heaps_evicted**

std::uint64_t &**incomplete_heaps_flushed**

std::uint64_t &**packets**

std::uint64_t &**batches**

std::uint64_t &**worker_blocked**

std::uint64_t &**max_batch**

std::uint64_t &**single_packet_heaps**

std::uint64_t &**search_dist**

class **stream_stat_config**

Registration information about a statistic counter.

Public Types

enum class **mode**

Type for a statistic.

All statistics are integral, but the mode determines how values are merged.

Values:

enumerator **COUNTER**

Merge values by addition.

enumerator **MAXIMUM**

Merge values by taking the larger one.

Public Functions

inline const std::string &**get_name**() const

Get the name passed to the constructor.

inline *mode* **get_mode**() const

Get the mode passed to the constructor.

std::uint64_t **combine**(std::uint64_t a, std::uint64_t b) const

Combine two samples according to the mode.

namespace **stream_stat_indices**

Constants for indexing *stream_stats* by index.

Variables

static constexpr std::size_t **heaps** = 0

static constexpr std::size_t **incomplete_heaps_evicted** = 1

static constexpr std::size_t **incomplete_heaps_flushed** = 2

static constexpr std::size_t **packets** = 3

static constexpr std::size_t **batches** = 4

static constexpr std::size_t **max_batch** = 5

static constexpr std::size_t **single_packet_heaps** = 6

static constexpr std::size_t **search_dist** = 7

static constexpr std::size_t **worker_blocked** = 8

static constexpr std::size_t **custom** = 9

Index for first user-defined statistic.

3.4 Sending

3.4.1 Heaps

class **heap**

Heap that is constructed for transmission.

Subclassed by spead2::send::heap_wrapper

Public Types

typedef std::vector<*item*>::size_type **item_handle**

Opaque handle type for retrieving previously added items.

Public Functions

explicit **heap**(const flavour &flavour_ = flavour())

Constructor.

Parameters

flavour_ – SPEAD flavour that will be used to encode the heap

inline const flavour &**get_flavour**() const

Return flavour.

template<typename ...**Args**>
inline *item_handle* **add_item**(s_item_pointer_t id, *Args*&&... args)

Construct a new item.

Returns

A handle that can be passed to *get_item* to update the item

inline *item* &**get_item**(*item_handle* handle)

Get a reference to a previously added item.

The retrieved item reference may be modified to update the heap in place. Behaviour is undefined if *handle* is not a handle previously returned by *add_item*.

Parameters

handle – Item handle previously returned from *add_item*

inline const *item* &**get_item**(*item_handle* handle) const

Get a reference to a previously added item.

Behaviour is undefined if *handle* is not a handle previously returned by *add_item*.

Parameters

handle – Item handle previously returned from *add_item*

inline void **add_pointer**(std::unique_ptr<std::uint8_t[]> &&pointer)

Take over ownership of *pointer* and arrange for it to be freed when the heap is freed.

void **add_descriptor**(const *descriptor* &descriptor)

Encode a descriptor to an item and add it to the heap.

inline void **add_start**()

Add a start-of-stream control item.

inline void **add_end**()

Add an end-of-stream control item.

inline void **set_repeat_pointers**(bool repeat)

Enable/disable repetition of item pointers in all packets.

Usually this is not needed, but it can enable some specialised use cases where immediates can be recovered from incomplete heaps or where the receiver examines the item pointers in each packet to decide how to

handle it. The packet size must be large enough to fit all the item pointers for the heap (the implementation also reserves a little space, so do not rely on a tight fit working).

The default is disabled.

inline bool **get_repeat_pointers**() const

Return the flag set by *set_repeat_pointers*.

struct **heap_reference**

Associate a heap with metadata needed to transmit it.

It holds a reference to the original heap.

Public Functions

inline **heap_reference**(const send::*heap* &heap, s_item_pointer_t cnt = -1, std::size_t substream_index = 0, double rate = -1.0)

Public Members

const send::*heap* &**heap**

s_item_pointer_t **cnt**

std::size_t **substream_index**

double **rate**

struct **item**

An item to be inserted into a heap.

An item does *not* own its memory.

Public Functions

item() = default

Default constructor.

This item has undefined values and is not usable.

inline **item**(s_item_pointer_t id, const void *ptr, std::size_t length, bool allow_immediate)

Create an item referencing existing memory.

inline **item**(s_item_pointer_t id, s_item_pointer_t immediate)

Create an item with a value to be encoded as an immediate.

inline **item**(s_item_pointer_t id, const std::string &value, bool allow_immediate)

Construct an item referencing the data in a string.

inline **item**(s_item_pointer_t id, const std::vector<std::uint8_t> &value, bool allow_immediate)

Construct an item referencing the data in a vector.

Public Members

s_item_pointer_t **id**

Item ID.

bool **is_inline**

If true, the item's value is stored in-place and *must* be encoded as an immediate.

Non-inline values can still be encoded as immediates if they have the right length.

bool **allow_immediate**

If true, the item's value may be encoded as an immediate.

This must be false if the item is variable-sized, because in that case the actual size can only be determined from address differences.

If *is_inline* is true, then this must be true as well.

const std::uint8_t ***ptr**

Pointer to the value.

std::size_t **length**

Length of the value.

s_item_pointer_t **immediate**

Integer value to store (host endian).

This is used if and only if *is_inline* is true.

3.4.2 Configuration

See `spead2.send.StreamConfig` for an explanation of the configuration options. In the C++ API, one must first construct a default configuration and then use setters to set individual properties. The setters all return the configuration itself so that one can construct a configuration with a single expression such as

```
spead2::send::stream_config().set_max_packet_size(9172).set_rate(1e9)
```

class **stream_config**

Configuration for send streams.

Public Functions

stream_config &**set_max_packet_size**(std::size_t max_packet_size)

Set maximum packet size to use (only counts the UDP payload, not L1-4 headers).

inline std::size_t **get_max_packet_size**() const

Get maximum packet size to use.

stream_config &**set_rate**(double rate)

Set maximum transmit rate to use, in bytes per second.

inline double **get_rate**() const

Get maximum transmit rate to use, in bytes per second.

stream_config &**set_burst_size**(std::size_t burst_size)

Set maximum size of a burst, in bytes.

inline std::size_t **get_burst_size**() const

Get maximum size of a burst, in bytes.

stream_config &**set_max_heaps**(std::size_t max_heaps)

Set maximum number of in-flight heaps.

inline std::size_t **get_max_heaps**() const

Get maximum number of in-flight heaps.

stream_config &**set_burst_rate_ratio**(double burst_rate_ratio)

Set maximum increase in transmit rate for catching up.

inline double **get_burst_rate_ratio**() const

Get maximum increase in transmit rate for catching up.

stream_config &**set_rate_method**(rate_method method)

Set rate-limiting method.

inline rate_method **get_rate_method**() const

Get rate-limiting method.

double **get_burst_rate**() const

Get product of rate and burst_rate_ratio.

3.4.3 Streams

All stream types are derived from `spead2::send::stream`.

enum class spead2::send::**group_mode**

Determines how to order packets when using *spead2::send::stream::async_send_heaps*.

Values:

enumerator **ROUND_ROBIN**

Interleave the packets of the heaps.

One packet is sent from each heap in turn (skipping those that have run out of packets).

enumerator **SERIAL**

Send the heaps serially, as if they were added one at a time.

typedef std::function<void(const boost::system::error_code &ec, item_pointer_t bytes_transferred)> spead2::send::*stream*::**completion_handler**

Callback type for asynchronous notification of heap completion.

class **stream**

Stream for sending heaps, potentially to multiple destinations.

Subclassed by *spead2::send::inproc_stream*, *spead2::send::streambuf_stream*, *spead2::send::tcp_stream*, *spead2::send::udp_ibv_stream*, *spead2::send::udp_stream*

Public Functions

boost::asio::io_service &**get_io_service**() const

Retrieve the io_service used for processing the stream.

void **set_cnt_sequence**(item_pointer_t next, item_pointer_t step)

Modify the linear sequence used to generate heap cnts.

The next heap will have cnt *next*, and each following cnt will be incremented by *step*. When using this, it is the user's responsibility to ensure that the generated values remain unique. The initial state is *next* = 1, *cnt* = 1.

This is useful when multiple senders will send heaps to the same receiver, and need to keep their heap cnts separate.

bool **async_send_heap**(const *heap* &h, *completion_handler* handler, s_item_pointer_t cnt = -1, std::size_t substream_index = 0, double rate = -1.0)

Send *h* asynchronously, with *handler* called on completion.

The caller must ensure that *h* remains valid (as well as any memory it points to) until *handler* is called.

If this function returns `true`, then the heap has been added to the queue. The completion handlers for such heaps are guaranteed to be called in order.

If this function returns `false`, the heap was rejected without being added to the queue. The handler is called as soon as possible (from a thread running the io_service). If the heap was rejected due to lack of space, the error code is `boost::asio::error::would_block`.

By default the heap cnt is chosen automatically (see *set_cnt_sequence*). An explicit value can instead be chosen by passing a non-negative value for *cnt*. When doing this, it is entirely the responsibility of the user to avoid collisions, both with other explicit values and with the automatic counter. This feature is useful when multiple senders contribute to a single stream and must keep their heap cnts disjoint, which the automatic assignment would not do.

The transmission rate may be overridden using the optional *rate* parameter. If it is negative, the stream's rate applies, if it is zero there is no rate limiting, and if it is positive it specifies the rate in bytes per second.

Some streams may contain multiple substreams, each with a different destination. In this case, *substream_index* selects the substream to use.

Return values

- **`false`** – If the heap was immediately discarded
- **`true`** – If the heap was enqueued

template<typename **CompletionToken**>
inline auto **async_send_heap**(const *heap* &h, *CompletionToken* &&token, s_item_pointer_t cnt = -1, std::enable_if_t<!std::is_convertible_v<*CompletionToken*, *completion_handler*>, std::size_t> substream_index = 0, double rate = -1.0)

Send *h* asynchronously, with an arbitrary completion token.

This overload is not used if the completion token is convertible to completion_handler.

Refer to the other overload for details. The boolean return of the other overload is absent. You will need to retrieve the asynchronous result and check for `boost::asio::error::would_block` to determine if the heaps were rejected due to lack of buffer space.

template<typename **Iterator**>

inline bool **async_send_heaps** (*Iterator* first, *Iterator* last, *completion_handler* handler, *group_mode* mode)

Send a group of heaps asynchronously, with *handler* called on completion.

The caller must ensure that the *heap* objects (as well as any memory they point to) remain valid until *handler* is called.

If this function returns `true`, then the heaps have been added to the queue. The completion handlers for such heaps are guaranteed to be called in order. Note that there is no individual per-heap feedback; the callback is called once to give the result of the entire group.

If this function returns `false`, the heaps were rejected without being added to the queue. The handler is called as soon as possible (from a thread running the io_service). If the heaps were rejected due to lack of space, the error code is `boost::asio::error::would_block`. It is an error to send an empty list of heaps.

Note that either all the heaps will be queued, or none will; in particular, there needs to be enough space in the queue for them all.

The heaps are specified by a range of input iterators. Typically they will be of type *heap_reference*, but other types can be used by overloading `get_heap`, `get_heap_cnt` and `get_heap_substream_index` for the value type of the iterator. Refer to *async_send_heap* for an explanation of the *cnt* and *substream_index* parameters.

The *heap_reference* objects can be safely deleted once this function returns; it is sufficient for the *heap* objects (and the data they reference) to persist.

Return values

- **`false`** – If the heaps were immediately discarded
- **`true`** – If the heaps were enqueued

template<typename **Iterator**, typename **CompletionToken**>
inline auto **async_send_heaps** (*Iterator* first, *Iterator* last, *CompletionToken* &&token, std::enable_if_t<!std::is_convertible_v<*CompletionToken*, *completion_handler*>, *group_mode*> mode)

Send a group of heaps asynchronously, with an arbitrary completion token (e.g., `boost::asio::use_future`).

This overload is not used if the completion token is convertible to completion_handler.

Refer to the other overload for details. There are a few differences:

- The boolean return of the other overload is absent. You will need to retrieve the asynchronous result and check for `boost::asio::error::would_block` to determine if the heaps were rejected due to lack of buffer space.
- Depending on the completion token, the iterators might not be used immediately. Using `boost::asio::use_future` causes them to be used immediately, but `boost::asio::deferred` or `boost::asio::use_awaitable` does not (they are only used when awaiting the result). If they are not used immediately, the caller must keep them valid (as well as the data they reference) until they are used.

std::size_t **get_num_substreams** () const

Get the number of substreams in this stream.

void **flush** ()

Block until all enqueued heaps have been sent.

This function is thread-safe; only the heaps that were enqueued prior to calling the function are waited for. The handlers will have been called prior to this function returning.

class **udp_stream** : public spead2::send::*stream*

Public Functions

udp_stream(*io_service_ref* io_service, const std::vector<boost::asio::ip::udp::endpoint> &endpoints, const *stream_config* &config = *stream_config*(), std::size_t buffer_size = default_buffer_size, const boost::asio::ip::address &interface_address = boost::asio::ip::address())

Constructor.

This constructor can handle unicast or multicast destinations, but is primarily intended for unicast as it does not provide all the options that the multicast-specific constructors do.

Parameters

- **io_service** – I/O service for sending data
- **endpoints** – Destination address and port for each substream
- **config** – Stream configuration
- **buffer_size** – Socket buffer size (0 for OS default)
- **interface_address** – Source address

  (see tips on *Routing*)

udp_stream(*io_service_ref* io_service, boost::asio::ip::udp::socket &&socket, const std::vector<boost::asio::ip::udp::endpoint> &endpoints, const *stream_config* &config = *stream_config*())

Constructor using an existing socket and an explicit io_service or thread pool.

The socket must be open but not connected, and the io_service must match the socket's.

udp_stream(*io_service_ref* io_service, const std::vector<boost::asio::ip::udp::endpoint> &endpoints, const *stream_config* &config, std::size_t buffer_size, int ttl)

Constructor with multicast hop count.

Parameters

- **io_service** – I/O service for sending data
- **endpoints** – Multicast group and port for each substream
- **config** – Stream configuration
- **buffer_size** – Socket buffer size (0 for OS default)
- **ttl** – Maximum number of hops

Throws

- `std::invalid_argument` – if any element of *endpoints* is not a multicast address
- `std::invalid_argument` – if the elements of *endpoints* do not all have the same protocol
- `std::invalid_argument` – if *endpoints* is empty

udp_stream (*io_service_ref* io_service, const std::vector<boost::asio::ip::udp::endpoint> &endpoints, const *stream_config* &config, std::size_t buffer_size, int ttl, const boost::asio::ip::address &interface_address)

Constructor with multicast hop count and outgoing interface address (IPv4 only).

Parameters

- **io_service** – I/O service for sending data
- **endpoints** – Multicast group and port for each substream
- **config** – Stream configuration
- **buffer_size** – Socket buffer size (0 for OS default)
- **ttl** – Maximum number of hops
- **interface_address** – Address of the outgoing interface

Throws

- `std::invalid_argument` – if any element of *endpoint* is not an IPv4 multicast address
- `std::invalid_argument` – if *endpoints* is empty
- `std::invalid_argument` – if *interface_address* is not an IPv4 address

udp_stream (*io_service_ref* io_service, const std::vector<boost::asio::ip::udp::endpoint> &endpoints, const *stream_config* &config, std::size_t buffer_size, int ttl, unsigned int interface_index)

Constructor with multicast hop count and outgoing interface index (IPv6 only).

See also:

if_nametoindex(3)

Parameters

- **io_service** – I/O service for sending data
- **endpoints** – Multicast group and port for each substream
- **config** – Stream configuration
- **buffer_size** – Socket buffer size (0 for OS default)
- **ttl** – Maximum number of hops
- **interface_index** – Index of the outgoing interface

Throws

- `std::invalid_argument` – if any element of *endpoints* is not an IPv6 multicast address
- `std::invalid_argument` – if *endpoints* is empty

Private Functions

udp_stream (*io_service_ref* io_service, boost::asio::ip::udp::socket &&socket, const std::vector<boost::asio::ip::udp::endpoint> &endpoints, const *stream_config* &config, std::size_t buffer_size)

Constructor used to implement most other constructors.

class **tcp_stream** : public spead2::send::*stream*

Public Functions

tcp_stream (*io_service_ref* io_service, std::function<void(const boost::system::error_code&)> &&connect_handler, const std::vector<boost::asio::ip::tcp::endpoint> &endpoints, const *stream_config* &config = *stream_config*(), std::size_t buffer_size = default_buffer_size, const boost::asio::ip::address &interface_address = boost::asio::ip::address())

Constructor.

A callback is provided to indicate when the connection is established.

> **Warning:** The callback may be called before the constructor returns. The implementation of the callback needs to be prepared to handle this case.

Parameters

- **io_service** – I/O service for sending data
- **connect_handler** – Callback when connection is established. It is called with a `boost::system::error_code` to indicate whether connection was successful.
- **endpoints** – Destination host and port (must contain exactly one element)
- **config** – Stream configuration
- **buffer_size** – Socket buffer size (0 for OS default)
- **interface_address** – Source address

  (see tips on *Routing*)

tcp_stream (*io_service_ref* io_service, boost::asio::ip::tcp::socket &&socket, const *stream_config* &config = *stream_config*())

Constructor using an existing socket.

The socket must be connected.

class **streambuf_stream** : public spead2::send::*stream*

Puts packets into a streambuf (which could come from an `ostream`).

This should not be used for a blocking stream such as a wrapper around TCP, because doing so will block the asio handler thread.

Subclassed by spead2::send::stream_wrapper< streambuf_stream >

Public Functions

streambuf_stream(*io_service_ref* io_service, std::streambuf &streambuf, const *stream_config* &config = *stream_config*())

Constructor.

3.5 In-process transport

See the *Python documentation* for an overview of the in-process transport.

class **inproc_queue**

Queue for packets being passed within the process.

While the data members are public, this is only to allow the send and receive code (and unit tests) to access the data. Users are advised to treat the data members as opaque.

Public Functions

void **add_packet**(packet &&pkt)

Add a packet directly to the queue.

void **stop**()

Indicate end-of-stream to receivers.

It is an error to add any more packets after this.

3.5.1 Sending

class **inproc_stream** : public spead2::send::*stream*

Public Functions

inproc_stream(*io_service_ref* io_service, const std::vector<std::shared_ptr<*inproc_queue*>> &queues, const *stream_config* &config = *stream_config*())

Constructor.

Private Functions

detail::queue_item ***get_queue**(std::size_t idx)

Access a (valid) item from the queue (takes care of masking the index)

3.5.2 Receiving

class **inproc_reader** : public spead2::recv::reader

Stream reader that receives packets from an *inproc_queue*.

Public Functions

inproc_reader (*stream* &owner, std::shared_ptr<*inproc_queue*> queue)

Constructor.

3.6 Logging

By default, log messages are all written to standard error. However, the logging function can be replaced by calling `spead2::set_log_function()`.

std::function<void(log_level, const std::string&)> spead2::**set_log_function** (std::function<void(log_level, const std::string&)>)

Set the callback used to display log messages, and return the old value.

3.7 Support for ibverbs

The support for libibverbs is essentially the same as for *Python*, with the same limitations. The programmatic interface is via the `spead2::recv::udp_ibv_reader` and `spead2::send::udp_ibv_stream` classes:

class **udp_ibv_config** : public spead2::detail::udp_ibv_config_base<*udp_ibv_config*>

Configuration for *udp_ibv_reader*.

Subclassed by spead2::recv::udp_ibv_config_wrapper

Public Functions

inline std::size_t **get_max_size** () const

Get maximum packet size to accept.

udp_ibv_config &**set_max_size** (std::size_t max_size)

Set maximum packet size to accept.

inline const std::vector<boost::asio::ip::udp::endpoint> &**get_endpoints** () const

Get the configured endpoints.

udp_ibv_config &**set_endpoints** (const std::vector<boost::asio::ip::udp::endpoint> &endpoints)

Set the endpoints (replacing any previous).

Throws

`std::invalid_argument` – if any element of *endpoints* is invalid.

udp_ibv_config &**add_endpoint**(const boost::asio::ip::udp::endpoint &endpoint)

Append a single endpoint.

Throws

`std::invalid_argument` – if *endpoint* is invalid.

inline const boost::asio::ip::address **get_interface_address**() const

Get the currently set interface address.

udp_ibv_config &**set_interface_address**(const boost::asio::ip::address &interface_address)

Set the interface address.

Throws

`std::invalid_argument` – if *interface_address* is not an IPv4 address.

inline std::size_t **get_buffer_size**() const

Get the currently configured buffer size.

udp_ibv_config &**set_buffer_size**(std::size_t buffer_size)

Set the buffer size.

The value 0 is special and resets it to the default. The actual buffer size used may be slightly different to round it to a whole number of packet-sized slots.

inline int **get_comp_vector**() const

Get the completion channel vector (see set_comp_vector)

udp_ibv_config &**set_comp_vector**(int comp_vector)

Set the completion channel vector (interrupt) for asynchronous operation.

Use a negative value to poll continuously. Polling should not be used if there are other users of the thread pool. If a non-negative value is provided, it is taken modulo the number of available completion vectors. This allows a number of streams to be assigned sequential completion vectors and have them load-balanced, without concern for the number available.

inline int **get_max_poll**() const

Get maximum number of times to poll in a row (see set_max_poll)

udp_ibv_config &**set_max_poll**(int max_poll)

Set maximum number of times to poll in a row.

If interrupts are enabled (default), it is the maximum number of times to poll before waiting for an interrupt; if they are disabled by set_comp_vector, it is the number of times to poll before letting other code run on the thread.

Throws

`std::invalid_argument` – if *max_poll* is zero.

Public Static Attributes

static constexpr std::size_t **default_buffer_size** = 16 * 1024 * 1024

Receive buffer size, if none is explicitly set.

static constexpr std::size_t **default_max_size** = udp_reader_base::default_max_size

Maximum packet size to accept, if none is explicitly set.

static constexpr int **default_max_poll** = 10

Number of times to poll in a row, if none is explicitly set.

class **udp_ibv_reader** : public spead2::recv::detail::udp_ibv_reader_base<*udp_ibv_reader*>

Synchronous or asynchronous stream reader that reads UDP packets using the Infiniband verbs API.

It currently only supports IPv4, with no fragmentation, IP header options, or VLAN tags.

Public Functions

udp_ibv_reader(*stream* &owner, const *udp_ibv_config* &config)

Constructor.

Parameters

- **owner** – Owning stream
- **config** – Configuration

Throws

- `std::invalid_argument` – If no endpoints are set.
- `std::invalid_argument` – If no interface address is set.

class **udp_ibv_config** : public spead2::detail::udp_ibv_config_base<*udp_ibv_config*>

Configuration for *udp_ibv_stream*.

Subclassed by spead2::send::udp_ibv_config_wrapper

Public Functions

inline std::uint8_t **get_ttl**() const

Get the IP TTL.

udp_ibv_config &**set_ttl**(std::uint8_t ttl)

Set the IP TTL.

inline const std::vector<memory_region> &**get_memory_regions**() const

Get currently registered memory regions.

udp_ibv_config &**set_memory_regions**(const std::vector<memory_region> &memory_regions)

Register a set of memory regions (replacing any previous).

Items stored inside such pre-registered memory regions can (in most cases) be transmitted without making a copy. A memory region is defined by a start pointer and a size in bytes.

Memory regions must not overlap; this is only validating when constructing the stream.

udp_ibv_config &**add_memory_region**(const void *ptr, std::size_t size)

Append a memory region (see *set_memory_regions*)

inline const std::vector<boost::asio::ip::udp::endpoint> &**get_endpoints**() const

Get the configured endpoints.

udp_ibv_config &**set_endpoints**(const std::vector<boost::asio::ip::udp::endpoint> &endpoints)

Set the endpoints (replacing any previous).

Throws
`std::invalid_argument` – if any element of *endpoints* is invalid.

udp_ibv_config &**add_endpoint**(const boost::asio::ip::udp::endpoint &endpoint)

Append a single endpoint.

Throws
`std::invalid_argument` – if *endpoint* is invalid.

inline const boost::asio::ip::address **get_interface_address**() const

Get the currently set interface address.

udp_ibv_config &**set_interface_address**(const boost::asio::ip::address &interface_address)

Set the interface address.

Throws
`std::invalid_argument` – if *interface_address* is not an IPv4 address.

inline std::size_t **get_buffer_size**() const

Get the currently configured buffer size.

udp_ibv_config &**set_buffer_size**(std::size_t buffer_size)

Set the buffer size.

The value 0 is special and resets it to the default. The actual buffer size used may be slightly different to round it to a whole number of packet-sized slots.

inline int **get_comp_vector**() const

Get the completion channel vector (see set_comp_vector)

udp_ibv_config &**set_comp_vector**(int comp_vector)

Set the completion channel vector (interrupt) for asynchronous operation.

Use a negative value to poll continuously. Polling should not be used if there are other users of the thread pool. If a non-negative value is provided, it is taken modulo the number of available completion vectors. This allows a number of streams to be assigned sequential completion vectors and have them load-balanced, without concern for the number available.

inline int **get_max_poll**() const

Get maximum number of times to poll in a row (see set_max_poll)

udp_ibv_config &**set_max_poll**(int max_poll)

Set maximum number of times to poll in a row.

If interrupts are enabled (default), it is the maximum number of times to poll before waiting for an interrupt; if they are disabled by set_comp_vector, it is the number of times to poll before letting other code run on the thread.

Throws
`std::invalid_argument` – if *max_poll* is zero.

Public Static Attributes

static constexpr std::size_t **default_buffer_size** = 512 * 1024

Default send buffer size.

static constexpr int **default_max_poll** = 10

Default number of times to poll in a row.

class **udp_ibv_stream** : public spead2::send::*stream*

Public Functions

udp_ibv_stream (*io_service_ref* io_service, const *stream_config* &config, const *udp_ibv_config* &ibv_config)

Constructor.

Parameters

- **io_service** – I/O service for sending data
- **config** – Common stream configuration
- **ibv_config** – Class-specific stream configuration

Throws

- `std::invalid_argument` – if *ibv_config* does not have an interface address set.
- `std::invalid_argument` – if *ibv_config* does not have any endpoints set.
- `std::invalid_argument` – if memory regions overlap.

3.7.1 PeerDirect

The pointer given to *spead2::send::udp_ibv_config::add_memory_region()* is passed to `ibv_reg_mr()`. When using an NVIDIA NIC, this can be a pointer that is handled by PeerDirect, such as a GPU device pointer. This can be used to transfer data directly from a GPU to the network without passing though the CPU.

This approach does need some care, because the spead2 implementation will fall back to copying if a packet contains too many discontiguous pieces of memory. It will be safe as long as there is only one item in a heap that uses a registered memory region, or as long as all such items are at least as big as the packet size.

For an example of this, see `examples/gpudirect_example.cu` in the spead2 source distribution.

3.8 Chunking receiver

For an overview, refer to *Chunking receiver*. This page is a reference for the C++ API.

struct **chunk_place_data**

Data passed to *chunk_place_function*.

This structure is designed to be a plain C structure that can easily be handled by language bindings. As far as possible, new fields will be added to the end but existing fields will be retained, to preserve ABI compatibility.

Do not modify any of the pointers in the structure.

Public Members

const std::uint8_t ***packet**

Pointer to the original packet data.

std::size_t **packet_size**

Number of bytes referenced by *packet*.

s_item_pointer_t ***items**

Values of requested item pointers.

std::int64_t **chunk_id**

Chunk ID (output). Set to -1 (or leave unmodified) to discard the heap.

std::size_t **heap_index**

Number of this heap within the chunk (output)

std::size_t **heap_offset**

Byte offset of this heap within the chunk payload (output)

std::uint64_t ***batch_stats**

Pointer to staging area for statistics.

std::uint8_t ***extra**

Pointer to data to be copied to *chunk::extra*.

std::size_t **extra_offset**

Offset within *chunk::extra* to write.

std::size_t **extra_size**

Number of bytes to copy to *chunk::extra*.

typedef std::function<void(*chunk_place_data* *data, std::size_t data_size)>
spead2::recv::**chunk_place_function**

Callback to determine where a heap is placed in the chunk stream.

See also:

chunk_place_data

Param data
Pointer to the input and output arguments.

Param data_size
`sizeof(chunk_place_data)` at the time spead2 was compiled

typedef std::function<std::unique_ptr<chunk>(std::int64_t chunk_id, std::uint64_t *batch_stats)>
`chunk_allocate_function`

Callback to obtain storage for a new chunk.

typedef std::function<void(std::unique_ptr<*chunk*>&&, std::uint64_t *batch_stats)>
`spead2::recv::`**`chunk_ready_function`**

Callback to receive a completed chunk.

It takes ownership of the chunk.

class **`chunk`**

Storage for a chunk with metadata.

Subclassed by spead2::recv::chunk_wrapper

Public Members

std::int64_t **`chunk_id`** = -1

Chunk ID.

std::uintptr_t **`stream_id`** = 0

Stream ID of the stream from which the chunk originated.

memory_allocator::pointer **`present`**

Flag array indicating which heaps have been received (one byte per heap)

std::size_t **`present_size`** = 0

Number of elements in *present*.

memory_allocator::pointer **`data`**

Chunk payload.

memory_allocator::pointer **`extra`**

Optional storage area for per-heap metadata.

class **chunk_stream_config**

Parameters for a *chunk_stream*.

Public Functions

chunk_stream_config &**set_items**(const std::vector<item_pointer_t> &item_ids)

Specify the items whose immediate values should be passed to the place function (see *chunk_place_function*).

inline const std::vector<item_pointer_t> &**get_items**() const

Get the items set with *set_items*.

chunk_stream_config &**set_max_chunks**(std::size_t max_chunks)

Set the maximum number of chunks that can be live at the same time.

A value of 1 means that heaps must be received in order: once a chunk is started, no heaps from a previous chunk will be accepted.

Throws

`std::invalid_argument` – if *max_chunks* is 0.

inline std::size_t **get_max_chunks**() const

Return the maximum number of chunks that can be live at the same time.

chunk_stream_config &**set_place**(*chunk_place_function* place)

Set the function used to determine the chunk of each heap and its placement within the chunk.

inline const *chunk_place_function* &**get_place**() const

Get the function used to determine the chunk of each heap and its placement within the chunk.

chunk_stream_config &**set_allocate**(*chunk_allocate_function* allocate)

Set the function used to allocate a chunk.

inline const *chunk_allocate_function* &**get_allocate**() const

Get the function used to allocate a chunk.

chunk_stream_config &**set_ready**(*chunk_ready_function* ready)

Set the function that is provided with completed chunks.

inline const *chunk_ready_function* &**get_ready**() const

Get the function that is provided with completed chunks.

chunk_stream_config &**enable_packet_presence**(std::size_t payload_size)

Enable the packet presence feature.

The payload offset of each packet is divided by *payload_size* and added to the heap index before indexing *spead2::recv::chunk::present*.

Throws

`std::invalid_argument` – if *payload_size* is zero.

chunk_stream_config &**disable_packet_presence**()

Disable the packet presence feature enabled by *enable_packet_presence*.

inline std::size_t **get_packet_presence_payload_size**() const

Retrieve the `payload_size` if packet presence is enabled, or 0 if not.

chunk_stream_config &**set_max_heap_extra**(std::size_t max_heap_extra)

Set maximum amount of data a placement function may write to *chunk_place_data::extra*.

inline std::size_t **get_max_heap_extra**() const

Get maximum amount of data a placement function may write to *chunk_place_data::extra*.

Public Static Attributes

static constexpr std::size_t **default_max_chunks** = 2

Default value for *set_max_chunks*.

class **chunk_stream** : private spead2::recv::detail::chunk_stream_state<detail::chunk_manager_simple>, public spead2::recv::*stream*

Stream that writes incoming heaps into chunks.

Subclassed by *spead2::recv::chunk_ring_stream< DataRingbuffer, FreeRingbuffer >*

Public Functions

chunk_stream(*io_service_ref* io_service, const *stream_config* &config, const *chunk_stream_config* &chunk_config)

Constructor.

This class passes a modified *config* to the base class constructor. This is reflected in the return of *get_config*. In particular:

- The *allow unsized heaps* setting is forced to false.
- The *memcpy function* may be overridden, although the provided function is still used when a copy happens. Use *get_heap_metadata* to get a pointer to heap_metadata, from which the chunk can be retrieved.
- The *memory allocator* is overridden, and the provided value is ignored.
- Additional statistics are registered:
  - `too_old_heaps`: number of heaps for which the placement function returned a non-negative chunk ID that was behind the window.
  - `rejected_heaps`: number of heaps for which the placement function returned a negative chunk ID.

Parameters

- **io_service** – I/O service (also used by the readers).
- **config** – Basic stream configuration
- **chunk_config** – Configuration for chunking

Throws

`invalid_argument` – if any of the function pointers in *chunk_config* have not been set.

Private Functions

inline const *chunk_stream_config* &**get_chunk_config**() const

Get the stream's chunk configuration.

Private Static Functions

static const heap_metadata ***get_heap_metadata**(const *memory_allocator*::pointer &ptr)

Get the metadata associated with a heap payload pointer.

If the pointer was not allocated by a chunk stream, returns `nullptr`.

3.8.1 Ringbuffer convenience API

template<typename **DataRingbuffer** = *ringbuffer*<std::unique_ptr<*chunk*>>, typename **FreeRingbuffer** = *ringbuffer*<std::unique_ptr<*chunk*>>>
class **chunk_ring_stream** : public detail::chunk_ring_pair<*ringbuffer*<std::unique_ptr<*chunk*>>, *ringbuffer*<std::unique_ptr<*chunk*>>>, public spead2::recv::*chunk_stream*

Wrapper around *chunk_stream* that uses ringbuffers to manage chunks.

When a fresh chunk is needed, it is retrieved from a ringbuffer of free chunks (the "free ring"). When a chunk is flushed, it is pushed to a "data ring". These may be shared between streams, but both will be stopped as soon as any of the streams using them are stopped. The intended use case is parallel streams that are started and stopped together.

When stop is called, any in-flight chunks (that are not in either of the ringbuffers) will be freed from the thread that called stop.

Public Functions

chunk_ring_stream(*io_service_ref* io_service, const *stream_config* &config, const *chunk_stream_config* &chunk_config, std::shared_ptr<*DataRingbuffer*> data_ring, std::shared_ptr<*FreeRingbuffer*> free_ring)

Constructor.

Refer to *chunk_stream::chunk_stream* for details.

The *allocate* callback is ignored and should be unset. If a *ready* callback is defined, it will be called before the chunk is pushed onto the ringbuffer. It must not move from the provided *unique_ptr*, but it can be used to perform further processing on the chunk before it is pushed.

Calling *get_chunk_config* will reflect the internally-defined callbacks.

template<typename **T**, typename **DataSemaphore** = semaphore, typename **SpaceSemaphore** = semaphore>

class **ringbuffer** : public spead2::ringbuffer_base<*T*>

Ring buffer with blocking and non-blocking push and pop.

It supports non-copyable objects using move semantics. The producer may signal that it has finished producing data by calling *stop*, which will gracefully shut down consumers as well as other producers. This class is fully thread-safe for multiple producers and consumers.

With multiple producers it is sometimes desirable to only stop the ringbuffer once all the producers are finished. To support this, a producer may register itself with *add_producer*, and indicate completion with *remove_producer*. If this causes the number of producers to fall to zero, the stream is stopped.

Public Functions

void **try_push**(*T* &&value)

Append an item to the queue, if there is space.

It uses move semantics, so on success, the original value is undefined.

Parameters
value – Value to move

Throws

- *ringbuffer_full* – if there is no space
- *ringbuffer_stopped* – if *stop* has already been called

template<typename ...**Args**>
void **try_emplace**(*Args*&&... args)

Construct a new item in the queue, if there is space.

Parameters
args – Arguments to the constructor

Throws

- *ringbuffer_full* – if there is no space
- *ringbuffer_stopped* – if *stop* has already been called

template<typename ...**SemArgs**>
void **push**(*T* &&value, *SemArgs*&&... sem_args)

Append an item to the queue, blocking if necessary.

It uses move semantics, so on success, the original value is undefined.

Parameters

- **value** – Value to move
- **sem_args** – Arbitrary arguments to pass to the space semaphore

Throws
ringbuffer_stopped – if *stop* is called first

template<typename ...**Args**>
void **emplace**(*Args*&&... args)

Construct a new item in the queue, blocking if necessary.

Parameters
args – Arguments to the constructor

Throws
ringbuffer_stopped – if *stop* is called first

T **try_pop**()

Retrieve an item from the queue, if there is one.

Throws

- *ringbuffer_stopped* – if the queue is empty and *stop* was called
- *ringbuffer_empty* – if the queue is empty but still active

template<typename ...**SemArgs**>

T **pop** (*SemArgs*&&... sem_args)

Retrieve an item from the queue, blocking until there is one or until the queue is stopped.

Parameters

sem_args – Arbitrary arguments to pass to the data semaphore

Throws

ringbuffer_stopped – if the queue is empty and *stop* was called

bool **stop** ()

Indicate that no more items will be produced.

This does not immediately stop consumers if there are still items in the queue; instead, consumers will continue to retrieve remaining items, and will only be signalled once the queue has drained.

Returns

whether the ringbuffer was stopped

bool **remove_producer** ()

Indicate that a producer registered with *add_producer* is finished with the ringbuffer.

If this was the last producer, the ringbuffer is stopped.

Returns

whether the ringbuffer was stopped

inline const *DataSemaphore* &**get_data_sem** () const

Get access to the data semaphore.

inline const *SpaceSemaphore* &**get_space_sem** () const

Get access to the free-space semaphore.

detail::ringbuffer_iterator<*ringbuffer*> **begin** ()

Begin iteration over the items in the ringbuffer.

This does not return a full-blown iterator; it is only intended to be used for a range-based for loop. For example: `for (auto &&item : ringbuffer) { ... }`

detail::ringbuffer_sentinel **end** ()

End iterator (see *begin*).

std::size_t **capacity** () const

Maximum number of items that can be held at once.

std::size_t **size** () const

Return the number of items currently in the ringbuffer.

This should only be used for metrics, not for control flow, as the result could be out of date by the time it is returned.

void **add_producer** ()

Register a new producer.

Producers only need to call this if they want to call *ringbuffer::remove_producer*.

class **ringbuffer_empty** : public std::runtime_error

Thrown when attempting to do a non-blocking pop from an empty ringbuffer.

class **ringbuffer_full** : public std::runtime_error

Thrown when attempting to do a non-blocking push to a full ringbuffer.

class **ringbuffer_stopped** : public std::runtime_error

Thrown when attempting to do a pop from an empty ringbuffer that has been stopped, or a push to a ringbuffer that has been stopped.

3.9 Chunking stream groups

For an overview, refer to *Chunking stream groups*. This page is a reference for the C++ API.

class **chunk_stream_group_config**

Configuration for *chunk_stream_group*.

Public Types

enum class **eviction_mode**

Eviction mode when it is necessary to advance the group window.

See the *overview* for more details.

Values:

enumerator **LOSSY**

force streams to release incomplete chunks

enumerator **LOSSLESS**

a chunk will only be marked ready when all streams have marked it ready

Public Functions

chunk_stream_group_config &**set_max_chunks**(std::size_t max_chunks)

Set the maximum number of chunks that can be live at the same time.

A value of 1 means that heaps must be received in order: once a chunk is started, no heaps from a previous chunk will be accepted.

Throws

`std::invalid_argument` – if *max_chunks* is 0.

inline std::size_t **get_max_chunks**() const

Return the maximum number of chunks that can be live at the same time.

chunk_stream_group_config &**set_eviction_mode**(*eviction_mode* eviction_mode_)

Set chunk eviction mode. See *eviction_mode*.

inline *eviction_mode* **get_eviction_mode**() const

Return the current eviction mode.

chunk_stream_group_config &**set_allocate**(*chunk_allocate_function* allocate)

Set the function used to allocate a chunk.

inline const *chunk_allocate_function* &**get_allocate**() const

Get the function used to allocate a chunk.

chunk_stream_group_config &**set_ready**(*chunk_ready_function* ready)

Set the function that is provided with completed chunks.

inline const *chunk_ready_function* &**get_ready**() const

Get the function that is provided with completed chunks.

Public Static Attributes

static constexpr std::size_t **default_max_chunks** = *chunk_stream_config*::*default_max_chunks*

Default value for *set_max_chunks*.

class **chunk_stream_group**

A holder for a collection of streams that share chunks.

The group owns the component streams, and takes care of stopping and destroying them when the group is stopped or destroyed.

It presents an interface similar to `std::vector` for observing the set of attached streams.

The public interface must only be called from one thread at a time, and all streams must be added before any readers are attached to them.

Subclassed by *spead2::recv::chunk_stream_ring_group< DataRingbuffer, FreeRingbuffer >*

Vector-like access to the streams.

Iterator invalidation rules are the same as for `std::vector` i.e., modifying the set of streams invalidates iterators.

inline std::size_t **size**() const

Number of streams.

inline bool **empty**() const

Whether there are any streams.

inline *chunk_stream_group_member* &**operator[]**(std::size_t index)

Get the stream at a given index.

inline const *chunk_stream_group_member* &**operator[]**(std::size_t index) const

Get the stream at a given index.

iterator **begin**() noexcept

Get an iterator to the first stream.

iterator **end**() noexcept

Get an iterator past the last stream.

const_iterator **begin**() const noexcept

Get an iterator to the first stream.

const_iterator **end**() const noexcept

Get a const iterator past the last stream.

const_iterator **cbegin**() const noexcept

Get an iterator to the first stream.

const_iterator **cend**() const noexcept

Get a const iterator past the last stream.

Public Functions

chunk_stream_group_member &**emplace_back**(*io_service_ref* io_service, const *stream_config* &config, const *chunk_stream_config* &chunk_config)

Add a new stream.

template<typename **T**, typename ...**Args**>
T &**emplace_back**(*Args*&&... args)

Add a new stream, possibly of a subclass.

virtual void **stop**()

Stop all streams and release all chunks.

class **chunk_stream_group_member** : private spead2::recv::detail::chunk_stream_state<detail::chunk_manager_group>, public spead2::recv::*stream*

Single single within a group managed by *chunk_stream_group*.

Public Functions

virtual void **stop_received**() override

Shut down the stream.

This calls flush_unlocked. Subclasses may override this to achieve additional effects, but must chain to the base implementation. It is guaranteed that it will only be called once.

It is undefined what happens if add_packet is called after a stream is stopped.

This is called with spead2::recv::stream_base::shared_state::queue_mutex locked. Users must not call this function themselves; instead, call *stop*.

virtual void **stop**() override

Stop the stream.

After this returns, the io_service may still have outstanding completion handlers, but they should be no-ops when they're called.

In most cases subclasses should override *stop_received* rather than this function. However, if heap_ready can block indefinitely, this function should be overridden to unblock it before calling the base implementation.

template<typename **T**, typename ...**Args**>
inline void **emplace_reader**(*Args*&&... args)

Add a new reader by passing its constructor arguments, excluding the initial *io_service* and *owner* arguments.

void **start**()

Start the stream.

This is only needed if the config specifies explicit start (see *stream_config::set_explicit_start*). In that case, no new readers can be added after starting the stream.

inline const *stream_config* &**get_config**() const

Get the stream's configuration.

stream_stats **get_stats**() const

Return statistics about the stream.

See the Python documentation.

Protected Functions

inline const *stream_config* &**get_config**() const

Get the stream's configuration.

stream_stats **get_stats**() const

Return statistics about the stream.

See the Python documentation.

void **flush**()

Flush the collection of live heaps, passing them to heap_ready.

Private Functions

stream_config **adjust_config**(const *stream_config* &config)

Compute the config to pass down to *spead2::recv::stream*.

std::pair<std::uint8_t*, heap_metadata> **allocate**(std::size_t size, const packet_header &packet)

Allocate storage for a heap within a chunk, given a first packet for a heap.

Returns

A raw pointer for heap storage and context used for actual copies.

void **flush_chunks**()

Send all in-flight chunks to the ready callback (not thread-safe)

inline const *chunk_stream_config* &**get_chunk_config**() const

Get the stream's chunk configuration.

Private Static Functions

static const heap_metadata ***get_heap_metadata**(const *memory_allocator*::pointer &ptr)

Get the metadata associated with a heap payload pointer.

If the pointer was not allocated by a chunk stream, returns `nullptr`.

3.9.1 Ringbuffer convenience API

template<typename **DataRingbuffer** = *ringbuffer*<std::unique_ptr<*chunk*>>, typename **FreeRingbuffer** = *ringbuffer*<std::unique_ptr<*chunk*>>>
class **chunk_stream_ring_group** : public detail::chunk_ring_pair<*ringbuffer*<std::unique_ptr<*chunk*>>, *ringbuffer*<std::unique_ptr<*chunk*>>>, public spead2::recv::*chunk_stream_group*

Wrapper around *chunk_stream_group* that uses ringbuffers to manage chunks.

When a fresh chunk is needed, it is retrieved from a ringbuffer of free chunks (the "free ring"). When a chunk is flushed, it is pushed to a "data ring". These may be shared between groups, but both will be stopped as soon as any of the members streams are stopped. The intended use case is parallel groups that are started and stopped together.

When the group is stopped, the ringbuffers are both stopped, and readied chunks are diverted into a graveyard. The graveyard is then emptied from the thread calling *stop*. This makes it safe to use chunks that can only safely be freed from the caller's thread (e.g. a Python thread holding the GIL).

Vector-like access to the streams.

Iterator invalidation rules are the same as for `std::vector` i.e., modifying the set of streams invalidates iterators.

inline std::size_t **size**() const

Number of streams.

inline bool **empty**() const

Whether there are any streams.

inline *chunk_stream_group_member* &**operator[]**(std::size_t index)

Get the stream at a given index.

inline const *chunk_stream_group_member* &**operator[]**(std::size_t index) const

Get the stream at a given index.

iterator **begin**() noexcept

Get an iterator to the first stream.

const_iterator **begin**() const noexcept

Get an iterator to the first stream.

iterator **end**() noexcept

Get an iterator past the last stream.

const_iterator **end**() const noexcept

Get a const iterator past the last stream.

const_iterator **cbegin**() const noexcept

Get an iterator to the first stream.

const_iterator **cend**() const noexcept

Get a const iterator past the last stream.

Public Functions

virtual void **stop**() override

Stop all streams and release all chunks.

chunk_stream_group_member &**emplace_back**(*io_service_ref* io_service, const *stream_config* &config, const *chunk_stream_config* &chunk_config)

Add a new stream.

template<typename **T**, typename ...**Args**>
T &**emplace_back**(*Args*&&... args)

Add a new stream, possibly of a subclass.

CHAPTER

FOUR

ADVANCED FEATURES

4.1 Chunking receiver

For some high-bandwidth use cases, dealing with heaps one at a time is not practical. For example, transferring data to an accelerator (such as a GPU) or another process may have a high overhead that needs to be amortised over many heaps. This is particularly true when using the Python API, as transferring a heap across the C++/Python boundary has a non-trivial overhead.

Furthermore, when grouping heaps together, it may be useful to assemble them according to their semantic position, rather than just in the order they arrived. For example, if the heaps represent samples in time, it is useful to place them within larger buffers according to their timestamps. The *chunking receiver* classes support these use cases.

The main limitation of these classes is that only the heap payload is accessible in the output. The heap items are used only to determine where each heap belongs and are then discarded, and descriptors are not accessible. It is also necessary for the receiver to know in advance how big the heaps will be. It is thus best suited to application-specific receivers that have prior knowledge of the heap layout.

Heaps are organised into *chunks*, which collect the payload from multiple heaps into a contiguous region of memory. There are assumed to be a continuous sequence of chunks, with consecutive 64-bit IDs. The user provides a callback (the *place callback*) which determines for each heap

- the chunk ID;
- the heap *index* within the chunk;
- the *offset* within the chunk.

The heap index is used to report which heaps are received, via a boolean array. The indices for a chunk should thus be consecutive integers starting from 0 (gaps are allowed, but will waste memory in the array). The offset is the byte offset within the storage for the chunk. The callback may also indicate that the heap should be ignored by returning a chunk ID of -1 (which is the value on entry, so this is the effect if the callback does not change the chunk ID).

Chunks are assumed to be received approximately in order (increasing ID) without gaps, but with a tolerance specified as a maximum number of contiguous chunks to have under construction at one time. If a heap is received whose chunk ID is too low, it is dropped.

When the first packet of the first heap of a chunk is seen, a user-provided callback (the *allocation callback*) is used to obtain the storage for the chunk. It is responsible for obtaining the memory for both the payload and the boolean array indicating the present heaps. It must also zero out the boolean array, and it may choose to fill the payload as well. Note that when an incomplete heap is received, it will not be marked present in the array, but some of the payload bytes may still have been written.

Chunks may also be requested from the callback even when no packets have been seen for them. This occurs when the new chunk ID is not contiguous with the previously seen chunk ID. Chunks are back-filled (up to the window size) so that they are present should an older heap arrive later. This can also happen for the very first packet of the stream, but

it is limited to chunks with non-negative IDs. Thus, if the first packet corresponds to chunk 0, there will not be any back-filling. It is worth noting that the chunk IDs used in the callback are strictly monotonic.

If the callback returns a null pointer, all work on this chunk is silently skipped. This is intended only for use in shutdown code (i.e., during the call to `stop`) to avoid needing to create chunks that will never be consumed.

Once a chunk is aged out (by the arrival of newer chunks), or when the stream is stopped, it is passed to another callback (the *ready callback*) for processing.

4.1.1 Packet presence

Instead of only getting information on which heaps were successfully received, it is possible to instead get information about which *packets* were received, even if some packets from a heap are missing. This is only possible if the amount of payload in each packet is known in advance. The payload offset item is divided by the expected payload size and added to the heap offset returned by the callback before being used.

When using this feature one may wish to enable the `allow_out_of_order` flag when configuring the stream, so that the loss of a packet in the middle of a heap does not prevent the following packets from being processed.

4.1.2 Extra data

In some cases the place callback may wish to extract additional data from each heap (such as immediate items) and make it available as part of the chunk. To do this:

1. Set the `extra` member of each chunk to a buffer to hold this data.
2. When configuring the stream, specify the maximum amount of data that may be written here per heap.
3. As part of the place callback, write data through the `extra` pointer, and set the `extra_offset` and `extra_size` fields to indicate how much data to copy to the chunk and at which byte offset within the `extra` field. The `extra_size` field defaults to zero, so if you do not have any extra data to emit these fields need not be explicitly set.

At present it is only possible to write a contiguous piece of data per heap.

The data is transferred to the chunk even if the heap is incomplete (and hence not marked in the `present` array).

4.1.3 Ringbuffer convenience API

A subclass is provided that takes care of the allocation and ready callbacks using ringbuffers of chunks (for Python, this is the only API provided). This is aimed at use with a fixed pool of chunks that is recycled. Two ringbuffers are used: one moves completed chunks from the stream to the consumer, and the other returns chunks that are no longer needed to the stream. It is strongly recommended that both ringbuffers have capacity that is equal to the maximum number of chunks in the system, so they they never fill up and block (each ringbuffer slot only requires space for a single pointer, so the cost is low).

While it is possible to add freed chunks directly to the free ringbuffer, a `spead2::recv::chunk_ring_stream::add_free_chunk()` convenience function takes care of some details. It zeros out the heap presence flags, and if the ringbuffer has been stopped, it fails silently rather than throwing an exception. This avoids the need for exception-handling code when the stream is being shut down.

The ringbuffers are passed to the stream constructor, and can be shared between streams. This provides a mechanism to have a shared pool of free chunks, or to multiplex chunks from several streams together to a single consumer. In the latter case, it is often necessary to know which stream produced the chunk. Set the *`stream ID`* when constructing each stream; it is available as an attribute of the corresponding chunks.

When the stream is stopped by the user, both ringbuffers are stopped too. This makes sharing ringbuffers appropriate only when the streams have the same lifetime. However (since version 3.6.0), if a stream is stopped due to network activity, the free ringbuffer is not stopped, and the data ringbuffer is only stopped if this was the last stream sharing the ringbuffer.

4.1.4 Examples

The spead2 source distribution includes a number of examples that use this API, in both C++ and Python.

4.1.5 Advice for senders

The ready callback uses items in the first received packet of each heap. It's thus critical that the first packet (and ideally, every packet) of the heap contains immediate items necessary for correctly placing the heap. Senders can ensure this by using `spead2.send.Heap.repeat_pointers`.

Item descriptors form part of the heap payload, and hence would get mixed up with the actual data in the payload. It is thus best to separate heaps into those that only have descriptors and those that only have data. One could also eliminate descriptors entirely, but they are quite useful for debugging. If descriptors are used, receivers must be prepared to ignore those heaps.

4.2 Chunking stream groups

While the *Chunking receiver* allows for high-bandwidth streams to be received with low overhead, it still has a fundamental scaling limitation: each chunk can only be constructed from a single thread. *Chunk stream groups* allow this overhead to be overcome, although not without caveats.

Each stream is still limited to a single thread. However, a *group* of streams can share the same sequence of chunks, with each stream contributing a subset of the data in each chunk. Making use of this feature requires that load balancing is implemented at the network level, using different destination addresses or ports so that the incoming heaps can be multiplexed into multiple streams.

As with a single chunk stream, the group keeps a sliding window of chunks and obtains new ones from an allocation callback. When the window slides forward, chunks that fall out the back of the window are provided to a ready callback. Each member stream also has its own sliding window, which can be smaller (but not larger) than the group's window. When the group's window slides forward, the streams' windows are adjusted to ensure they still fit within the group's window. In other words, a stream's window determines how much reordering is tolerated within a stream, while the group's window determines how out-of-sync the streams are allowed to become.

When desynchronisation does occur, there is a choice of strategies. The default strategy is eager but potentially lossy: when the group's window moves forward, the trailing chunk is marked ready as soon as possible, even if this causes some stream windows to shrink below their normal size. An alternative strategy is lossless: when the group's window needs to move forward, it is blocked until all the member streams have caught up. This latter mode is intended for use with lossless transports such as TCP. However, if one of the component streams stops functioning (for example, because it is routed on a network path that is down) it prevents the entire group from making forward progress.

The general flow (in C++) is

1. Create a `chunk_stream_group_config`.
2. Create a `chunk_stream_group`.
3. Use `chunk_stream_group::emplace_back()` to create the streams.
4. Add readers to the streams.
5. Process the data.

6. Optionally, call `chunk_stream_group::stop()` (otherwise it will be called on destruction).
7. Destroy the group.

In Python the process is similar, although garbage collection replaces explicit destruction.

4.2.1 Ringbuffer convenience API

As for standalone chunk streams, there is a simplified API using ringbuffers, which is also the only API available for Python. A `chunk_stream_ring_group` is a group that allocates data from one ringbuffer and send ready data to another. The description of *that api* largely applies here too. The ringbuffers can be shared between groups.

4.2.2 Caveats

This is an advanced API that sacrifices some user-friendliness for performance, and thus some care is needed to use it safely.

- It is vital that all the streams can make forward progress independently, as otherwise deadlocks can occur. For example, if they share a thread pool, the pool must have at least as many threads as streams. It's recommended that each stream has its own single-threaded thread pool.
- The streams must all be added to the group before adding any readers to the streams. Once a group has received some data, an exception will be thrown if one attempts to add a new stream.
- The stream ID associated with each chunk will be the stream ID of one of the component streams, but it is undefined which one.
- When the allocate and ready callbacks are invoked, it's not specified which stream's batch statistics pointer will be passed.
- Two streams must not write to the same bytes of a chunk (in the payload, present array or extra data), as this is undefined behaviour in C++.
- Calling `stop()` on a member stream will stop the whole group.

4.3 Receiver stream statistics

A receive stream can be queried for statistics about the packets and heaps of the stream. Note that while the statistics below are expected to be stable except where otherwise noted, their exact interpretation in edge cases is subject to change as the implementation evolves. It is intended for instrumentation, rather than for driving application logic.

Each time the statistics are queried, an internally consistent view is returned. However, it is not synchronised with other aspects of the stream. For example, it's theoretically possible to retrieve 5 heaps from the stream iterator, then find that the `heaps` statistic is (briefly) 4. Querying the statistics is somewhat expensive, so if multiple statistics are needed, it is advisable to assign the result to a local variable first.

Some readers process packets in batches, and the statistics are only updated after a whole batch is added. This can be particularly noticeable if the ringbuffer fills up and blocks the reader, as this prevents the batch from completing and so heaps that have already been received by user code might not be reflected in the statistics.

Statistics can be accessed in several ways:

1. A map/dictionary-like interface, by name. This includes iteration with standard Python or C++ iterator conventions.
2. A list-like interface, by index. This is more efficient, so may be useful if statistics are being accessed very frequently, but less convenient. To determine the index for a particular statistic, use `StreamConfig.get_stat_index()` (Python) or `spead2::recv::stream_config::get_stat_index()` (C++).

3. By attribute/field access. This is for backwards compatibility and is only available for the core statistics. New code should prefer the other interfaces.

4.3.1 Core statistics

heaps
Total number of heaps put into the stream. This includes incomplete heaps, and complete heaps that were received but did not make it into the ringbuffer before `stop()` was called. It excludes the heap that contained the stop item.

incomplete_heaps_evicted
Number of incomplete heaps that were evicted from the buffer to make room for new data.

incomplete_heaps_flushed
Number of incomplete heaps that were still in the buffer when the stream stopped.

packets
Total number of packets received, including the one containing the stop item.

batches
Number of batches of packets. Some readers are able to take multiple packets from the network in one go, and each time this forms a batch.

worker_blocked
Number of times a worker thread was blocked because the ringbuffer was full. If this is non-zero, it indicates that the stream is not being read fast enough, or that the *heaps* constructor parameter to the `RingStreamConfig` needs to be increased to buffer sudden bursts.

In C++ this statistic is always present, but is only used by `spead2::recv::ring_stream`.

max_batch
Maximum number of packets received as a unit. This is only applicable to readers that support fetching a batch of packets from the source.

single_packet_heaps
Number of heaps that were entirely contained in a single packet. These take a slightly faster path as it is not necessary to reassemble them.

search_dist
Number of hash table entries searched to find the heaps associated with packets. This is intended for debugging/profiling spead2 and **may be removed without notice**.

4.3.2 Chunk receiver statistics

These statistics are only present when using a *chunking receiver*.

too_old_heaps
Heaps for which the chunk placement function returned a non-negative chunk ID, but one which was too old to be accepted (behind the moving window).

rejected_heaps
Heaps for which the chunk placement function returned a negative chunk ID to indicate that the heap should be discarded.

4.3.3 Custom statistics

It may be convenient for user code to collect additional statistics to be made available through the existing statistics framework. The framework takes care of thread-safe transfer of statistics from the worker threads that run the networking to the thread that is querying the statistics. However, this means that these custom statistics can *only* be safely updated from these worker threads. Some examples of places where it is safe to do so:

1. In the `heap_ready` virtual function.
2. In a *custom memory allocator* (when called to allocate memory for a heap).
3. In a *custom memory scatter* function.
4. In a chunk placement callback (see *Chunking receiver*).

All but the last are currently available in the C++ API only. They should use `spead2::recv::stream::batch_stats`. For the chunk placement callback, a pointer to the batch statistics is available in the `spead2::recv::chunk_place_data` structure.

As the name implies, this provides access only to statistics collected for a batch of packets received at the same time. At the end of the batch, the long-term statistics for the stream are updated from these batch statistics. The manner in which this update occurs depends on the *mode* of the statistic, which is one of the following:

counter
A count of events. The batch value is added to the long-term value.

maximum
A high water mark. The long-term value is set to the maximum of the previous value and the batch value.

The mode is set when registering the statistic with the stream config (`StreamConfig.add_stat()` or `spead2::recv::stream_config::add_stat()`).

Registration also returns the “index” of the statistic, which is used when accessing the batch statistics array. If many statistics are being registered, it may be inconvenient to keep track of all their indices. The index is guaranteed to increase by one with each registration, so one can instead record just the first index, and then compute other indices from it as needed.

Since all statistics (custom and core) share a single namespace, it is recommended that you prefix your custom statistics with a package name and a dot (`mypackage.mystatistic`) to ensure that they do not conflict with future statistics added by spead2. It’s also recommended to stick to printable ASCII for maximum compatibility across language bindings.

CHAPTER

FIVE

PERFORMANCE TUNING

While spead2 tries to be performant out of the box, there are a number of ways one can tune both the system and the application using spead2. It is usually necessary to do at least some of these steps to achieve performance of 10Gb/s+, but your mileage may vary depending on your hardware and application.

This guide focuses mostly on the problem of receiving data, because my experience with high-bandwidth SPEAD has been with data produced by FPGAs. Nevertheless, some of these tips also apply to sending data.

All advice is for a GNU/Linux system with an x86-64 CPU. You will need to consult other documentation to find equivalent commands for other systems.

5.1 System tuning

5.1.1 Kernel bypass networking

For the best performance it is necessary to bypass the kernel networking stack. At present the only kernel bypass technology supported by spead2 is *ibverbs*. Refer to that documentation for setup and tuning instructions. Note that at present this is only known to work with NVIDIA NICs.

5.1.2 Kernel network stack

If you're unable to bypass the kernel networking stack, this section has some advice on tuning it. The first thing to do is to increase the maximum socket buffer sizes. See *Introduction to spead2* for details.

The kernel firewall can affect performance, particularly if small packets are being used (in this context, anything that isn't a jumbo frame is considered "small"). If possible, remove all firewall rules and unload the kernel modules (those prefixed with `ipt` or `nf`). In particular, simply having the `nf_conntrack` module loaded can reduce performance by several percent.

IP fragmentation also causes performance problems on the receiver. Check that the routers in your network have a sufficiently large MTU that packets do not get fragmented, particularly if using jumbo frames. You can use **`tcpdump -v`** to see fragments.

The above all applies to UDP. For TCP, dropped packets are far less of a concern, and overly large buffers may actually be counter-productive as they do not fit in cache and lead to buffer-bloat. The simplest way to improve performance is to increase the packet size in the sender.

Routing

The constructors for the TCP and UDP stream classes take an *interface_address* argument, but the actual behaviour depends on the constructor used and the OS. For multicast constructors, it sets the `IP_MULTICAST_IF` socket option and hence directly determines the interface used. In other cases, it only determines the source IP address (via *bind(2)*), and the effect depends on the operating system. On Linux, the routing table determines the interface, so this setting will only impact routing if you have policy routing configured.

5.1.3 CPU

On a system with multiple CPU sockets, it is important to pin the process using spead2 to a single socket, so that memory accesses do not cross the inter-socket bus. For best performance, use the same socket as the NIC, which can be determined from the output of **hwloc-ls**. See *numactl(8)*, *hwloc-ls(1)*, *hwloc-bind(1)*.

There are a number of settings that can be adjusted to improve the system's ability to respond to bursts of data. These will probably not improve peak performance, but can reduce the number of lost heaps, particularly when a stream starts and the system must ramp up performance in response.

- Disable hyperthreading.
- Disable CPU frequency scaling.
- Disable C states beyond C1 (for example, by passing `processor.max_cstate=1` to the Linux kernel). Disabling C1 as well may reduce latency, but will likely limit the gains from Turbo Boost.
- Investigate disabling the P-state driver by passing `intel_pstate=disable` on the kernel command line. The P-state driver has sometimes been reported to be much slower[1],[2], but can also be faster[3].
- Disable adaptive interrupt moderation on the NIC: `ethtool -C` *`interface`* `adaptive-rx off adaptive-tx off`. You may then need to experiment to tune the interrupt moderation settings — consult *ethtool(8)* for details (does not apply if using ibverbs).
- Disable Ethernet flow control: `ethtool -A` *`interface`* `rx off tx off`.
- Use the isolcpus kernel option to completely isolate some CPU cores from other tasks, and pin the receiver to those cores (I have not actually tried this).
- Use *chrt(1)* to run the receiver with real-time scheduling (I have not actually tried this).

AMD Epyc tuning

There are also some specific BIOS settings that are important for AMD Epyc systems (these have been tried on Rome and Milan systems):

- Set APBDIS to 1 and Fixed SOC PState to P0. This prevents the bus from going into low-power states when it thinks there isn't enough work.
- Disable DF Cstates.
- Enable PCIe relaxed ordering.
- Experiment to find the best NUMA-per-socket setting. NPS4 gives slightly higher throughput but it also seems to let GPUs starve the NIC.
- When placing cards into slots, be aware that slots that connect to the same quadrant of the CPU (NUMA node, in NPS4 mode) contend for bandwidth to the CPU.

[1] https://www.phoronix.com/scan.php?page=article&item=intel_pstate_linux315
[2] https://www.phoronix.com/scan.php?page=article&item=linux-47-schedutil
[3] https://www.phoronix.com/scan.php?page=news_item&px=Linux-4.4-CPUFreq-P-State-Gov

The above have all been observed to make significant differences in spead2 applications. Below are some other general tuning recommendations found on the internet, for which it's unclear whether it will make a difference:

- Disable IOMMU.
- Set local APIC mode to x2APIC.
- Set Preferred I/O to manual and Preferred I/O bus to the bus containing the NIC (only really useful for a single NIC).
- On Milan, set LCLK frequency control to the maximum frequency.

Interrupt affinity

NICs typically have multiple send and receive queues with their own interrupt numbers, and each interrupt is typically directed to a particular CPU core. This means that not all CPU cores are equal when it comes to pinning threads. Generally you want the receiver to run "close" to the core handling the interrupts, so that the interrupt handler can wake it up easily, and driver data structures can be cached; but if they are on the same core it can sometimes reduce performance by contending for resources.

The drivers for NVIDIA NICs include some tools (**`show_irq_affinity.sh`**, **`set_irq_affinity.sh`**) to show and set IRQ affinities, which can help to set the affinities in a predictable way. Note that for this to be effective, the **`irqbalance`** daemon needs to be disabled, as it will try to dynamically adjust IRQ affinities based on usage patterns.

5.2 Protocol design

If you are designing a new SPEAD-based protocol, you have an opportunity to make design choices that will make it easier for the sender and/or receiver to reach the desired performance.

5.2.1 Heap size

The primary influence comes from heap size. There is some degree of overhead for every heap (particularly for a Python receiver), and very small heaps will cause this overhead to dominate. Heaps smaller than 16KiB are not recommended. Very large heaps that do not fit into CPU caches will also reduce performance, but not excessively. Memory usage also depends on the heap size. A number of application tuning techniques described below also depend on knowing the heap payload size a priori; thus, it is good practice to communicate this the receiver in some way, whether by sending the descriptor early in the SPEAD stream or by an out-of-band method.

5.2.2 Packet size

Packet size is not strictly part of the protocol, but also has a large impact on performance. For 10Gb/s or faster streams, jumbo frames are highly recommended, although with the kernel bypass techniques described below, this is far less of an issue.

When using spead2 on the send side, the default packet size is 1472 bytes, which is a safe value for IPv4 in a standard Ethernet setup[4]. The packet size is set in the *`StreamConfig`*. You should pick a packet size, that, when added to the overhead for IP and UDP headers, does not exceed the MTU of the link. For example, with IPv4 and an MTU of 9200, use a packet size of 9172.

When using TCP/IP, the packet size can be much larger (e.g. 65536) as it no longer corresponds to IP packets.

[4] The UDP and IP header together add 28 bytes, bringing the IP packet to the conventional MTU of 1500 bytes.

5.2.3 Alignment

Because items directly reference the received data (where possible), it is possible that data will be misaligned. While numpy allows this, it could make access to the data inefficient. The sender should ensure that data are aligned. The spead2 sending API currently does not provide a way to enforce this, but using items with round sizes will help.

5.2.4 Endianness

When using numpy builtin types, data are converted to native endian when they are received, to allow for more efficient operations on them. This can reduce the maximum rate at which packets are received. Thus, using the native endian on the wire (little-endian for x86) will give better performance.

5.2.5 Data format

Item descriptors can be specified using either a *format* or a *dtype* (numpy data type). In many common cases, either can be used, and performance on a Python receiver should be the same (a PySPEAD receiver, however, will be much faster with *dtype*). The *dtype* is the only way to use Fortran order or little-endian. The *format* approach is easier for a C++ receiver to parse (since it does not need to decode a Python literal). It also allows for a wider variety of types (such as bit vectors), but encoding or decoding these types in Python takes a very slow path.

5.3 Application tuning

This section describes a number of ways the application can be modified to improve performance. Most of these tuning options can be explored using a provided *benchmarking tool* which measures the sustained performance on a connection. This makes it possible to quickly identify the techniques that will make the most difference before implementing them.

5.3.1 Memory allocation

Using a *memory pool* is the single most important tool for fast and reliable data transfer. It is particularly important when heap sizes are large enough that `malloc()` and `free()` use `mmap()` (`M_MMAP_THRESHOLD` in glibc). For very small heaps, memory pooling may be a net loss.

To use a memory pool, it is necessary to know the maximum heap payload size (a conservative estimate is fine too — you will just use more memory). You also need to size the pool appropriately. It is possible to specify a small initial size and a larger maximum; however, each time the pool grows the CPU will be busy with allocation and may drop packets. To avoid starvation, you will need to provide:

- A buffer per partial heap (*max_heaps* parameter to `spead2.recv.Stream`)
- A buffer per complete heap in the ring buffer (*ring_heaps* parameter to `spead2.recv.Stream`)
- A buffer for every heap that has been taken off the ring buffer but not yet destroyed.
- A few extra for heaps that are in-flight between queues. The exact number may vary between releases, but 4 should be safe.

In general, it is best to err on the side of adding a few extra, provided that this does not consume too much memory. At present there are unfortunately no good tools for analysing memory pool performance.

Chunking receiver

An alternative to using memory pools is to use the *Chunking receiver*. It has the same benefit of keeping memory allocated within the application rather than returning it to the OS.

Heap lifetime (Python)

All the payload for a heap is stored in a single memory allocation, and where possible, items reference this memory. This means that the entire heap remains live as long as any of the values encoded in it are live. Thus, a small but seldom-changing value can cause a very large heap to remain live long after the rest of the values in that heap have been replaced. This can waste memory, and also affects memory pool sizing.

To avoid this, senders should try to group items together that are updated at the same frequency, rather than mixing low- and high-frequency items in the same heap. Receivers can avoid this problem by copying values that are known to be slowly varying.

Custom allocators

If you are doing an extra copy purely to put values into a special memory type (for example, shared memory to communicate with another process, or pinned memory for transfer to a GPU), then consider subclassing `spead2::memory_allocator` (C++ only), or using a *Chunking receiver*.

5.3.2 Tuning based on heap size

The library has a number of tuning parameters that are reasonable for medium-to-large heaps (megabytes or larger). If using many smaller heaps, some of the tuning parameters may need to be adjusted. In particular

- Increase the *max_heaps* parameter to the `spead2.send.StreamConfig` constructor.
- Increase the *max_heaps* parameter to the `spead2.recv.Stream` constructor if you expect the network to reorder packets significantly (e.g., because data is arriving from multiple senders which are not completely synchronised). For single-packet heaps this has no effect.
- Increase the *ring_heaps* parameter to the `spead2.recv.Stream` constructor to reduce lock contention. This has rapidly diminishing returns beyond about 16.

It is important to experiment to determine good values. Simply cranking everything way up can actually reduce performance by increase memory usage and thus reducing cache efficiency.

For very large heaps (gigabytes) some of these values can be decreased to 2 (or possibly even 1) to keep memory usage under control.

5.3.3 Thread pools

Each stream in spead2 has an associated thread pool, which provides worker threads for handling incoming or outgoing packets. Each thread pool can have some number of threads, defaulting to 1. Here are some rules of thumb:

- For a small number of streams (up to about the number of CPU cores), it is best to have one single-threaded thread pool per stream. This gives better cache affinity than a shared thread pool.
- For a large number of lower-bandwidth streams, use a shared thread pool with multiple threads. The number of threads should be chosen based on the number of CPU cores that you can dedicate to packet handling rather than other tasks in your application.

- A single stream cannot be processed by multiple threads at the same time, so there is never any benefit (and often detriment) to have more threads in a thread pool than there are streams serviced by that thread pool.
- Jitter (experienced as occasionally lost heaps) can be reduced by passing an affinity list to the thread pool constructor, to pin threads to specific cores. The main thread can be pinned as well, using `spead2.ThreadPool.set_affinity()`.

CHAPTER

SIX

COMMAND-LINE TOOLS

6.1 spead2_bench

A benchmarking tool is provided to estimate the maximum throughput for UDP. There are two versions: one implemented in Python 3 (**spead2_bench.py**) and one in C++ (**spead2_bench**), which are installed by the corresponding installers. The examples show the Python version, but the C++ version functions very similarly. However, they cannot be mixed: use the same version on each end of the connection.

On the receiver, pick a port number (which must be free for both TCP and UDP) and run

```
spead2_bench.py agent <port>
```

Then, on the sender, run

```
spead2_bench.py master [options] <host> <port>
```

where *host* is the hostname of the receiver. This script will run tests at a variety of speeds to determine the maximum speed at which the connection seems reliable most of the time. This speed is right at the edge of stability: for a totally reliable setup, you should use a lower speed.

6.2 spead2_send/spead2_recv

There are also separate **spead2_send** and **spead2_recv** (and Python 3 equivalent) programs. The former generates a stream of meaningless data, while the latter consumes an existing stream and reports the heaps and items that it finds. Apart from being useful for debugging a stream, **spead2_recv** has a similar plethora of command-line options for tuning that allow for exploration.

6.3 mcdump

mcdump is a tool similar to tcpdump, but specialised for high-speed capture of UDP traffic using hardware that supports the Infiniband Verbs API. It has only been tested on NVIDIA ConnectX NICs. Like gulp, it uses a separate thread for disk I/O and CPU core affinity to achieve reliable performance. With a sufficiently fast disk subsystem, it is able to capture line rate from a 40Gb/s adapter.

It is not limited to capturing SPEAD data. It is included with spead2 rather than released separately because it reuses a lot of the spead2 code.

6.3.1 Installation

The tool is automatically compiled and installed with spead2, provided that libibverbs support is detected at configure time.

It may also be necessary to configure the system to work with ibverbs. See *Support for ibverbs* for more information.

6.3.2 Usage

The simplest incantation is

```
mcdump -i xx.xx.xx.xx output.pcap yy.yy.yy.yy:zzzz
```

which will capture on the interface with IP address *xx.xx.xx.xx*, for the multicast group *yy.yy.yy.yy* on UDP port *zzzz*. mcdump will take care of subscribing to the multicast group. Note that only IPv4 is supported. Capture continues until interrupted by `Ctrl-C`. You can also list more `group:port` pairs, which will all stored in the same pcap file.

While originally written for multicast, mcdump also supports unicast. An IP address must still be provided; usually it will be the same as the interface address, but it could be a different address if the interface has multiple IP addresses.

You can also specify - in place of the filename to suppress the write to file. This is useful to simply count the bytes/packets received without being limited by disk throughput.

Unfortunately, unlike tcpdump, it is not possible to directly tell whether packets were dropped. NIC counters (on Linux, accessed with **ethtool -S**) can give an indication, although sometimes packets are dropped during the shutdown process.

These options are important for performance:

-N <cpu>, **-C** <cpu>, **-D** <cpu>
: Set CPU core IDs for various threads. The *-D* option can be repeated multiple times to use multiple threads for disk I/O. By default, the threads are not bound to any particular core. It is recommended that these cores be on the same CPU socket as the NIC.

--direct-io
: Use the `O_DIRECT` flag to open the file. This bypasses the kernel page cache, and can in some cases yield higher performance. However, not all filesystems support it, and it can also reduce performance when capturing a small enough amount of data that it will fit into RAM.

--count <count>
: Stop after <count> packets have been received. Without this option, mcdump will run until SIGINT (Ctrl-C) is received.

6.3.3 Limitations

- Only IPv4 is supported.
- It is not optimised for small packets (below about 1KB). Packet capture rates top out around 6Mpps for current hardware.

6.4 spead2_net_raw

When using *ibverbs*, it is necessary to have the CAP_NET_RAW capability on Linux. While this can be achieved by running as root, doing so may be undesirable. The **spead2_net_raw** utility program can be used to simplify running ibverbs applications. To use it, the program must first be given the capability. After installation, this can be done by running

```
sudo setcap cap_net_raw+p /usr/local/bin/spead2_net_raw
```

Adjust the path as necessary to match your installation. If **spead2_net_raw** did not get installed, check that you have the libcap development headers installed (for example, libcap-dev in Ubuntu), and rerun **configure** to detect it.

Now you can prefix any command with **spead2_net_raw** and it will have the CAP_NET_RAW capability. It is an "ambient" capability, so all child processes will have the capability too, which can be useful if the process you run is a shell.

> **Warning:** After doing the above, any user on the system that can run **spead2_net_raw** will be able to intercept any incoming network traffic or generate arbitrary outgoing traffic. You should not do this blindly if there are untrusted users on your system, or if the system allows untrusted code to run outside of a secure sandbox.

This is not the only way to give **spead2_net_raw** the capability (you can, for example, make it an "inherited" capability), but a full discussion of the Linux capabilities model is beyond the scope of this manual.

CHAPTER

SEVEN

MIGRATING TO VERSION 3

Version 3 makes a number of breaking changes, for the purpose of keeping the number of constructor arguments under control and making future extension more manageable. Almost all code will need to be updated, but the updates will in most cases be minor.

To allow for code that wishes to support both version 3 and older versions, C++ macros are defined to allow the version number to be interrogated at compile time. Thus, version 3 can be detected as

```
#if defined(SPEAD2_MAJOR) && SPEAD2_MAJOR >= 3
// Version 3 or later
#else
// Older
#endif
```

Note that version 3.0.0b1 did not define these macros, but also did not include the breaking changes.

SPEAD2_MAJOR

Major version of spead2 e.g., `3` for version 3.4.6.

SPEAD2_MINOR

Minor version of spead2 e.g., `4` for version 3.4.6.

SPEAD2_PATCH

Patch level of spead2 e.g., `6` for version 3.4.6.

SPEAD2_VERSION

Full spead2 version number, as a string constant.

In Python, one can get the full version string from `spead2.__version__`. Use the classes in `packaging.version` to analyse it.

7.1 Receive stream configuration

Prior to version 3, some parameters to configure a stream were passed directly to the constructor (e.g., the maximum number of partial heaps), while others were set by methods after construction (such as the memory allocator). In version 3, all these parameters are set at construction time, and they are held in helper classes `spead2.recv.StreamConfig` and `spead2.recv.RingStreamConfig` (`spead2::recv::stream_config` and `spead2::recv::ring_stream_config` for C++). Code will need to be modified to construct these helper objects.

Similarly, for *ibverbs* there is now a `spread.recv.UdpIbvConfig` that is used to configure the reader. The old forms of `spead.recv.Stream.add_udp_ibv_reader()` are still present but are deprecated.

In version 2 it was also possible (although not recommended) to change parameters like the memory allocator after readers had already been placed. For efficiency reasons this is no longer supported in version 3.

7.2 Send stream configuration

The changes for sending are more minor: the constructor for the Python class *spead2.send.StreamConfig* now only takes keyword arguments, and the C++ equivalent `spead2::send::stream_config` takes no constructor arguments. To make it convenient to construct temporaries, the setter methods return the object, allowing configurations to be constructed in a "fluent" style e.g.:

```
spead2::send::stream_config().set_max_packet_size(9172).set_rate(1e6)
```

For *ibverbs* streams the changes are more significant. There is now a *spead2.send.UdpIbvConfig* class that works similarly to *spead2.send.StreamConfig*, but configures properties specific to the ibverbs stream. The old constructor is still available (but deprecated); however, the constants `UdpIbvStream.DEFAULT_BUFFER_SIZE` and `UdpIbvStream.DEFAULT_MAX_POLL` have moved to the *UdpIbvConfig* class.

7.3 Substreams

A new feature is the ability to create a send stream with multiple destinations and select the destination on a per-heap basis (see *Substreams* for more information). Supporting this cleanly required a number of changes:

- The `spead2.send.InprocStream.queue` attribute has been replaced with *queues*. Similarly, the C++ *spead2::send::inproc_stream::get_queue()* has been replaced by `get_queues()`. The originals are still present but deprecated, and raise a `RuntimeError` if the stream was constructed with multiple queues.
- The constructors for most send stream types now accept a list of endpoints (or queues) rather than a single endpoint (queue). The old constructors are still supported for backwards compatibility, but are deprecated.
- The **spead2_send** and **spead2_send.py** example programs now take the destination in the form *host*:*port* instead of *host port*, and support multiple destinations.

7.4 Out-of-order packets

In prior versions of spead2, the packets forming a single heap could be received in any order. Starting with version 3, the default is to assume that packets arrive in order. Refer to *Packet ordering* for more details.

7.5 Loop argument to asyncio functions

The Python asyncio-based classes and functions no longer take a *loop* argument. As of Python 3.6, `asyncio.get_event_loop()` returns the executing event loop, so there is no need to pass the loop explicitly.

7.6 Command-line arguments in tools

The command-line handling in **spead2_send**, **spead2_recv** and **spead2_bench** has been overhauled and made more consistent. For example, **spead_bench** now supports the `--ttl` option, and `--send-ibv` is now an argument-less flag with the interface address given by `--send-bind` (and similarly for receive). See the help for each command for details of the current options.

7.7 Removal of deprecated functionality

The following functions were deprecated in version 2 and have been removed in version 3:

- C++ stream constructors that specified a socket but not an `io_service` (they could not be supported with Boost 1.70 onwards).
- Stream constructors that took both an existing (but unconnected) socket and a buffer size or a port to bind to. The caller should instead bind the socket (if receiving) and set any desired buffer size socket option.

7.8 Queue depth for sending with ibverbs

When using *ibverbs* to send data, heaps were previously considered complete once the packets were submitted to the hardware. They're now only considered complete once the hardware has indicated completion, which allows for errors to be reported. While there are no breaking API changes, if the heaps are very small it may be necessary to increase *max_heaps* in `StreamConfig` so that enough heaps can be in flight to fully utilise the buffer.

CHAPTER

EIGHT

MIGRATING TO VERSION 4

Unlike version 3, version 4 does not make substantial changes to the spead2 API, although it does drop deprecated functionality. It replaces the build system and increases the minimum version requirements, which may have implications for how you install, link to or use spead2.

8.1 Removed functionality

The following deprecated functionality has been removed:

8.1.1 C++

Functionality	Replacement
`recv::udp_ibv_reader::default_buffer_size`	*recv::udp_ibv_config::default_buffer_size*
`recv::udp_ibv_reader::default_max_poll`	*recv::udp_ibv_config::default_max_poll*
`send::udp_ibv_stream::default_buffer_size`	*send::udp_ibv_config::default_buffer_size*
`send::udp_ibv_stream::default_max_poll`	*send::udp_ibv_config::default_max_poll*
recv::udp_ibv_reader constructors that do not take a *recv::udp_ibv_config*	constructor that takes a *recv::udp_ibv_config*
send::udp_ibv_stream constructor that does not take a *send::udp_ibv_config*	constructor that takes a *send::udp_ibv_config*
send::inproc_stream constructor taking a single queue	pass a vector containing a single queue
`send::inproc_stream::get_queue()`	`send::inproc_stream::get_queues()[0]`
send::tcp_stream and *send::udp_stream* constructors taking a single endpoint	pass a vector containing a single endpoint

8.1.2 Python

recv.Stream.DEFAULT_UDP_IBV_BUFFER_SIZE	recv.UdpIbvConfig.DEFAULT_BUFFER_SIZE
recv.Stream.DEFAULT_UDP_IBV_MAX_SIZE	recv.UdpIbvConfig.DEFAULT_MAX_SIZE
recv.Stream.DEFAULT_UDP_IBV_MAX_POLL	recv.UdpIbvConfig.DEFAULT_MAX_POLL
send.UdpIbvStream.DEFAULT_BUFFER_SIZE	send.UdpIbvConfig.DEFAULT_BUFFER_SIZE
send.UdpIbvStream.DEFAULT_MAX_POLL	send.UdpIbvConfig.DEFAULT_MAX_POLL
recv.Stream.add_udp_ibv_reader() overload that does not take a *recv.UdpIbvConfig*	Pass a *recv.UdpIbvConfig*
send.UdpIbvStream constructors that do not take a *send.UdpIbvConfig*	Pass a *send.UdpIbvConfig*
send.InprocStream constructor taking a single queue	Pass a list containing a single queue
send.InprocStream.queue	*send.InprocStream.queues*[0]
send.TcpStream and *send.UdpStream* constructors taking a single hostname and port	Pass a list containing a single (*host*, *port*) tuple

8.2 Meson

The autotools build system has been replaced by *Meson*. This mainly affects C++ users, as for Python this is hidden behind the Python packaging interface. Refer to the *Introduction* for installation instructions.

The old build system had a number of options to adjust the build. The table below shows corresponding Meson options:

autotools	meson
`--enable-debug-symbols`	`debug=true` or `buildtype=...`
`--enable-debug-log`	`max_log_level=debug`
`--enable-coverage`	`b_coverage=true`
`--disable-optimized`	`optimization=0` or `buildtype=debug`
`--enable-lto`	`b_lto=true`
`--enable-shared`	`default_library=both`
`--without-program-options`	`tools=disabled`
`--without-ibv`	`ibv=disabled`
`--without-mlx5dv`	`mlx5dv=disabled`
`--without-ibv-hw-rate-limit`	`ibv_hw_rate_limit=disabled`
`--without-pcap`	`pcap=disabled`
`--without-cap`	`cap=disabled`
`--without-recvmmsg`	`recvmmsg=disabled`
`--without-sendmmsg`	`sendmmsg=disabled`
`--without-eventfd`	`eventfd=disabled`
`--without-posix-semaphores`	`posix_semaphores=disabled`
`--without-pthread_setaffinity_np`	`pthread_setaffinity_np=disabled`
`--without-fmv`	`fmv=disabled`
`--without-movntdq`	`sse2_stream=disabled`
`--without-cuda`	`cuda=disabled`
`--without-gdrapi`	`gdrapi=disabled`

Link-time optimization no longer requires intervention to select suitable versions of **`ar`** and **`ranlib`**; Meson takes care of it.

8.3 C++17

The codebase now uses C++17, whereas older versions used C++11. This might require a newer C++ compiler. See the *Introduction* for minimum compiler versions.

Additionally, when compiling against the C++ API, you may need to pass compiler arguments to select at least C++17 (e.g. `--std=c++17`). GCC 11+ and Clang 16+ support C++17 without a compiler flag, but keep in mind that your users might use older compilers.

8.4 Boost

Boost 1.69+ is now required: from this release, boost_system is a header-only library. You no longer need to link against any Boost libraries when linking against spead2.

8.5 pcap

The detection logic for libpcap has changed. It used to first try **pkg-config**, then fall back to testing compilation. It now tries **pkg-config** first and falls back to **pcap-config**. If neither of those methods works, you may need to upgrade your pcap library.

8.6 Code generation

In older versions of spead2, some of the code was generated and included in the release tarballs. If you used a release, you would be unaware of this, but trying to build directly from git would require you to run a `bootstrap.sh` script.

Meson doesn't have good support for including generated code into releases, so these generated files are no longer included in the releases, and they are instead created as part of the build. This requires Python, with the `jinja2`, `pycparser` and `packaging` packaging installed.

An advantage of this approach is that it is now possible to directly build from a git checkout without any preparatory steps.

8.7 Python configuration

When building the Python bindings from source, it was previously only possible to adjust the build-time configuration by editing source files. With the new build system, it's now possible to pass options on the command line.

8.8 Python editable installs

Meson-python doesn't support editable installs with build isolation. To make an editable install, use `pip install --no-build-isolation -e .`.

CHAPTER NINE

DEVELOPER DOCUMENTATION

9.1 Development processes

9.1.1 Getting started with development

Python setup

Refer to the *Introduction to spead2* for the prerequisite packages (particularly *Python install from source*). You will also need ninja-build, and a Python virtual environment. You can use any tool you like to create the virtual environment, but it must be located outside of the working copy of the spead2 repository[1]. I like pyenv (with pyenv-virtualenv), but you can also use venv or virtualenv directly. It's also a good idea to install ccache or sccache, as it will make recompilation much faster.

Check out a copy of spead2 from git, and make it your current directory. Install pre-commit (e.g., with **pip install pre-commit**) and run **pre-commit install** to set it up. This will ensure that commits pass a set of static analysis checks.

Inside your virtual environment, install the build dependencies. You can find a list of them right at the top of `pyproject.toml`.

You're now ready to make an "editable" installation. This is an installation that will use the files inside your working copy (recompiling C++ code if necessary), so that you don't need to explicitly install after each change. To do so, run

```
pip install --no-build-isolation -e .
```

See the meson-python documentation for more information about the limitations of editable installs. Also install the development runtime dependencies:

```
pip install -r requirements.txt       # If Python < 3.12
pip install -r requirements-3.12.txt  # If Python >= 3.12
```

Now you should be able to run the unit tests by executing **pytest**. It is expected that some tests will be skipped, because they require specific hardware. Running **pytest -ra** will show the reasons for skipped tests. You should expect to see something like:

```
SKIPPED [23] tests/test_passthrough.py:577: Envar SPEAD2_TEST_IBV_INTERFACE_ADDRESS␣
↪not set
```

If you're running the latest version of Python, it's possible that `numba` does not yet support it and so was not installed above. In that case you will have additional skipped tests e.g.:

[1] Meson will show a long error starting with "ERROR: Tried to form an absolute path to a dir in the source tree." There is also a Meson bug that causes this error to appear if the source directory is a prefix *as a string* of the virtual environment path, even if the virtual environment is not inside the source directory.

```
SKIPPED [1] tests/test_recv_chunk_stream.py:30: could not import 'numba': No module named 'numba'
SKIPPED [1] tests/test_recv_chunk_stream_group.py:30: could not import 'numba': No module named 'numba'
```

MacOS will have some additional skipped tests. On Linux, there should be no other skipped tests if you have all the optional dependencies installed. If there are other tests skipped, it is not a show-stopper; it just means you'll need to rely on the CI to run those tests for you.

C++ setup

You should start by following the steps for Python. Most of the functionality is only tested via the Python unit tests, so you will need to be able to run those even if you are only interested in working on the C++ bindings.

You can then follow the *Installing spead2 for C++* instructions to build the C++ bindings (you can skip **meson install**). From the build directory, also run **meson test** to run the C++ unit tests. These are a small set of tests that cover functionality that is not practical to test from the Python API.

Documentation

To install the necessary Python requirements, run **pip install -r requirements-readthedocs.txt**. You will also need doxygen and **make**. Then change to the `doc` directory and run **make**. This will build documentation in `doc/_build/html`. It is unfortunately normal for there to be a large number of warnings about duplicates.

Coding style

The first rule is just to adhere the existing style. Python code uses black and ruff to enforce style, so if you deviate from the style those tools will guide you back on track. The Python code generally does not use inline type annotations, because annotations in the `.pyi` files take precedence (and spead2 pre-dates Python 3 annotation syntax). New code (particularly in tests) can be annotated, but it is not required.

Identifiers use US English spelling, but comments, log messages and documentation favour UK spelling.

The C++ code is less consistent in style, but here are some guidelines:

- Use 4 spaces for indentation (**never** tabs).
- Opening braces go on their own line (Allman style). An exception is that a function may be written entirely on one line if it is very short.
- Do not use trailing commas.
- Do not add a level of indentation inside namespaces.
- When two levels of namespaces start and end at the same point, use the C++17 nested namespace syntax:

```
namespace spead2::recv
{
/* Stuff */
} // namespace spead2::recv
```

- When closing a namespace or a `#endif`, use a comment to indicate what is being closed, unless it is visually obvious (nearby and without further nesting).
- Be sparing with using `auto` to declare local variables. It should ideally be possible for the user to guess what the type is just by inspecting the code. Good reasons to use `auto` include:

  - The type is impossible to specify safely, because it is a lambda, or an implementation-defined type that could change in future.
  - It is an integer type, and explicitly naming the type could inadvertently cause type conversions if the type of the expression later changed.
  - The type is obvious from the initialiser, such as

```
auto foo = std::make_unique<Foo>(1);
```

  - The type is exceedingly long to write out (iterator types are a good example).

- Start a class with friends, followed by typedefs, member variables, and finally member functions. Put private members before public ones, unless a specific order is required (for example, to optimise memory layout or to control initialisation/destruction order).
- Line comments (`//`) should only be used for one-line comments (maybe two at a push). Use block comments (`/* */`) for longer blocks of text.
- If a member function has an empty body and exists only to implement a concept, it can use anonymous parameters if they are self-explanatory. Otherwise, unused parameters should be named but have the `[[maybe_unused]]` attribute. In some cases a particular compiler may still generate warnings after applying the attribute (GCC 9 has been seen to do this); in such cases one should place the parameter name inside `/* */`.

Committing

Before committing, remember to run **`pre-commit install`** to set up pre-commit. One of the pre-commit hooks checks that the requirements files are up to date, and (at the time of writing) depends on having both **`python3.8`** and **`python3.12`** commands on the path. If you're not touching the requirements, you can skip this hook by setting the environment variable `SKIP=pip-compile` when committing.

Making a pull request

spead2 uses the normal Github workflow for pull requests. There are many guides on the internet to writing good pull requests, such as this one or this one. A few points to note for spead2:

- Don't add to the changelog. The changelog for each release is generally prepared just prior to each release. However, it is a good idea to write a meaningful title for the pull request that could become the changelog entry.
- Once a pull request has been reviewed, don't force-push changes. Doing so prevents the reviewer from seeing the difference between the previously-reviewed version and your update. If you're a stickler for a neat commit history, ask if you can rebase just prior to merging.

9.1.2 Repository layout

C++

`src/*.cpp`
: Source files. Files are grouped into functionality by prefixes:

  `src/recv_*.cpp`
  : Receiving data (`spead2::recv` namespace).

  `src/send_*.cpp`
  : Sending data (`spead2::send` namespace).

src/common_*.cpp
Other general shared code (`spead` namespace).

src/unittest_*.cpp
C++ unit tests.

src/spead2_*.cpp and src/mcdump.cpp
Command-line utilities.

src/py_*.cpp
Python bindings.

src/*.h
Header files that are only used internally (not installed for users).

include/spead/*.h
Header files that are installed and form the public API. The filenames mostly correspond to the source files.

examples/*.cpp
Example code.

Python

src/spead2/
Source code. This is placed within a `src` subdirectory so that Python does not automatically import from it unless explicitly added to the Python path. See Packaging a Python Library for an explanation of the advantages.

src/spead2/tools/
Implementations of the command-line tools.

examples/*.py
Example code.

tests/
Unit tests. These are mainly for use with pytest, but `tests/shutdown.py` contains tests that are run to ensure that the interpret shuts down cleanly (see *Interpreter shutdown*).

Other

gen/
Utilities that run as part of the build.

doc/
Documentation.

9.1.3 Debugging

Debug builds

Meson provides standard infrastructure for doing debug builds. Specifically, these disable optimisation and enable assertions. For C++ builds you can pass `--buildtype=debug` when setting up the build directory (note that Meson supports multiple build directories, so you can keep separate directories for release and debug builds if you like). For Python builds, you can pass `-Csetup-args=--buildtype=debug` to **pip install**. Note that for an editable install, this option is *sticky*: invoking **pip install** in future without this option will not reset it to the default, unless you delete the `build` directory.

Debug logging

Debug builds do not automatically enable debug-level logging. See the *Logging* documentation for instructions to do that.

Debug symbols for Python wheels

Occasionally a bug may manifest in a released Python wheel but prove impossible to reproduce with a locally-compiled version of the package. While it will not give a great debugging experience (because the code is optimised), it is possible to install separate debug symbols so that one can get line numbers from stack traces. Note that this is only supported on Linux.

On the Github page for the release is a file called `spead2-`*`version`*`-debug.tar.xz`. Unpack it into `lib/pythonX.X/site-packages/spead2` inside your virtual environment). You only need to install the `.debug` file matching the Python version and architecture. It should have the same name as an existing file in the same directory, but with the `.debug` suffix. Once this is done, GDB should be able to load the debug symbols from this file. Note that it will only work for the released wheel from the same version; if you compile a wheel yourself then the build ID will most likely not match and GDB will not use it.

Reducing worker threads

Numpy creates a lot of worker threads, which can make it more difficult to find the thread of interest in gdb. Setting the environment variable `OMP_NUM_THREADS` to 1 will reduce the number of threads to sift through.

9.1.4 Release checklist

- Update the version number and changelog in `doc/changelog.rst`
- Update the version number in `VERSION.txt`
- Update the shared library version number in `meson.build`:
  - If there are ABI changes, update the first number and reset the second to zero.
  - Otherwise, increment the second number.
- Check that `.pyi` stubs have been updated
- Check that Github Actions successfully tested the release and built wheels
- Install the sdist from Github Actions and check that it passes pytest
- Install a wheel from Github Actions and check that it passes pytest
- Tag the release
- Run **`git push --tags`**
- Upload the sdist and wheels to PyPI with twine
- Upload the sdist and debug symbols to Github release
- Check that readthedocs has updated itself

9.2 Design

This section documents internal design decisions that users will generally not need to be aware of, although some of it may be useful if you plan to subclass the C++ classes to extend functionality.

9.2.1 Python bindings

The Python bindings are implemented using pybind11, which handles most of the work of exposing C++ classes and members to Python.

Global Interpreter Lock

The Python Global Interpreter Lock (GIL) is both good news and bad news for the bindings. The good news is that it ensures that (by default) only one Python thread can be calling the spead2 API at a time, which provides some basic thread safety. Here's the bad news (but also read the pybind11 documentation on the topic):

- Without additional steps, blocking calls to the spead2 API would prevent other Python threads from making progress. Mostly this just hurts performance, but it can even lead to deadlocks. This is mostly handled by judicious use of `pybind11::gil_scoped_release` in functions that wait for events or take C++ locks.
- When a thread does not hold the GIL (either because it released it, or because it is a C++ thread created internally) it cannot safely call Python APIs. While `pybind11::object` is a very convenient interface for reference counting, it can cause accidental use of the reference counting APIs through destructors or copy constructors. For any class that embeds a Python object, it is important to know from which threads it could be destroyed (and in some cases the core spead2 design has had to be modified to account for this). Some pieces of code that do not expect to do any reference counting choose to use `pybind11::handle` instead to eliminate the risk.
- If a worker thread tries to acquire the GIL, this could block for a significant period of time, which in turn could cause high latency for time-critical work (such as processing packets).

  Logging to the Python logging system is one area where this used to be problematic. That has been solved by putting log messages (as C++ strings) on a ringbuffer and using a dedicated worker thread to pass the log messages to Python. If this worker thread is blocked, log messages simply accumulate in the ringbuffer, and eventually log messages get dropped, but the thread that is doing the logging is not blocked.

  A similar issue is notifying Python code that some asynchronous work has completed. This is covered in a later section.

Note that once **PEP 703** is implemented, the spead2 bindings may need significant rework to support no-GIL mode.

Semaphores

spead2 uses semaphores extensively for producer/consumer relationships between threads. Dropping the GIL before calling a blocking function will partially help, but has two potential issues:

1. If the blocking function waits on a semaphore and then takes action that uses the Python API, it will need to re-acquire the GIL *before* taking that action.
2. If the user presses Ctrl-C, it will not interrupt the program: it will interrupt the low-level semaphore wait (with `EINTR`), but the default semaphore implementation in spead2 will simply retry that wait. Graceful handling of Ctrl-C requires using `PyErr_CheckSignals()` to give Python a chance to handle the interrupt.

To address both concerns, several functions (particularly in *ringbuffer*) take a variadic number of extra arguments, which are passed on to `semaphore_get()`. The Python bindings provide an overload of `semaphore_get()`, selected by passing an instance of the empty class `gil_release_tag`, that handles both releasing the GIL and checking `PyErr_CheckSignals()` each time the semaphore wait is interrupted.

Asynchronous callbacks

The C++ interface for sending heaps asynchronously takes a callback which is executed when the heap has been transmitted. While the high-level Python interface exposed to users wraps this into an `asyncio.Future`, the low-level interface exposed from C++ to Python still takes a callback. However, the callback is Python code and can only be called with the GIL held. Furthermore, since `asyncio` is not thread-safe, it should be run in the thread that's running the asyncio event loop, rather than the C++ thread that's doing the networking.

To solve these problems, the C++-level callback doesn't directly invoke the Python callback. Instead, it puts it in a vector of deferred callbacks. To wake up the event loop without locking the GIL, it writes data to a file descriptor (eventfd in Linux, but it's abstracted by `semaphore_fd`), which the asyncio event loop watches. The event loop thread then requests the stream to execute all its deferred callbacks.

As an optimisation, the file descriptor is only touched if the callback list was empty. Thus, a rapid burst of complete heaps only requires one wakeup. This requires some careful use of a mutex to correctly handle the case where new callbacks are added while the callback list is being processed.

Interpreter shutdown

If spead2 objects are still present when the Python interpreter is shut down, their destructors may try to interact with the Python API after it is too late to safely do so. For example, the logging system may try to interact with the GIL after the GIL has already been destroyed. Thus, some objects need to be shut down earlier in the interpreter shutdown process, and this is achieved using the `atexit` module. An `exit_stopper` class simplifies the process.

9.2.2 Locking and asio for receive

Spead2 uses a somewhat unusual combination of locking with asynchronous I/O. It causes a few problems and ideally should be redesigned one day. First, let's introduce some terminology: the user writes code which runs on a thread, which we'll call the "user thread" (there may be multiple user threads, but for most functions it's only safe for one user thread at a time to interact with a stream). The threads in a *ThreadPool*, or more generally threads running `boost::asio::io_service::run()` (or equivalents) are "worker threads".

The original sin that leads to many complications is the back-pressure mechanism for receiving: if a ring-buffer is full, then pushing a heap (or chunk) to it simply blocks the worker thread, rather than signalling to readers that they should stop listening for data until space becomes available. Blocking a worker thread is generally a bad thing to do in asynchronous programming, and if not handled carefully can lead to deadlocks. Even when it is safe, it can lead to inefficiencies since the blocked thread is sitting idle when there could be other work for it. This is one reason that sharing thread pools between streams is not recommended (another is cache locality). Fixing this would require major and backwards-incompatible redesign to allow for control-flow signalling.

Locking is needed for a few reasons:

- The user thread and a worker thread may need to access the same data. Version 3 reduced the number of places this can happen by making most of the configuration immutable, but it is still needed to stop the stream and to access statistics.
- In a stream with multiple readers and multiple worker threads, it is possible for multiple worker threads to need access to the stream internals concurrently.

Early versions of spead2 solved these problems using strands, where functions invoked by the user thread would post work to the strand and use futures to return the result to the user thread. This lead to many issues with deadlocks, and debugging was difficult because this control flow was not apparent in the call stack. It may be worth revisiting now that there are fewer places where the user thread needs to interact with the stream internals, but it will be necessary to compare the performance to the locking approach.

Batching

Even in the absence of contention, locking can be expensive, and we found that taking and releasing a lock for every packet had a significant cost. The design was thus changed to ensure that multiple packets can be handled with a single lock. This complements APIs such as `recvmmsg()` that allow multiple available packets to be retrieved at once.

This batching approach is realised by the `spead2::recv::stream::add_packet_state` class. Constructing the class takes a lock on the stream, and the destructor releases it. This class also holds local statistics for the batch, which are used to update the stream-wide statistics at the end of the batch.

Unfortunately, pushing completed heaps to the user is done with this lock held, which means that not only is the worker thread blocked if the ringbuffer is full, but any other thread (including a user thread) that needs the lock will also be blocked.

Stopping

There are four circumstances under which a receive stream can stop:

1. A stream control heap is received from the network.
2. A transport-level event occurs, such as a connection being closed by the remote end.
3. The user calls `stream::stop()`.
4. The user destroys the stream (which implicitly calls `stream::stop()`).

The first two are referred to as "network stops" and the latter two as "user stops". Both cases involve call `stream::stop_received()`, but only user stops invoke `stream::stop()`.

A fundamental difference between the two cases is that for network stops, the user is generally waiting for data from the stream, and so one can assume that the ringbuffer will generally empty out in finite time. What's more, the user may wish to actually receive all the data that was transmitted prior to the network stop. With user stops, the user is generally not consuming from the ringbuffer, and it must be possible to stop the stream even if the ringbuffer is full, even if this means losing data that is still arriving from the network.

To handle user stops correctly, stream classes whose `stream_base::heap_ready()` function potentially blocks must override `stop()` to unblock it. Classes that use ringbuffers (`ring_stream`, `chunk_ring_stream` etc.) do so by stopping the ringbuffer *before* calling the base class implementation. This causes any blocked (and future) attempts to push data into the ringbuffer to immediately fail with an exception. This does mean that some data that was received is dropped. On the other hand, network stops do *not* immediately stop the ringbuffer, and allow any data still in the stream to be flushed. This does mean that if there is no consumer for the ringbuffer, the worker thread could be blocked until the user stop (or resumes consuming from the ringbuffer).

9.2.3 Destruction of receive streams

The asynchronous and parallel nature of spead2 makes destroying a receive stream a tricky operation: there may be pending asio completion handlers that will try to push packets into the stream, leading to a race condition. While asio guarantees that closing a socket will cancel any pending asynchronous operations on that socket, this doesn't account for cases where the operation has already completed but the completion handler is either pending or is currently running.

Up to version 3.11, this was handled by a shutdown protocol between `stream` and `reader`. The reader was required to notify the stream when it had completely shut down, and `stream::stop()` would block until all readers had performed this notification (via a semaphore). This protocol was complicated, and it relied on the reader being able to make forward progress while the thread calling `stream::stop()` was blocked.

Newer versions take a different approach based on shared pointers. The ideal case would be to have the whole stream always managed by a shared pointer, so that a completion handler that interfaces with the stream could keep a copy of the shared pointer and thus keep it alive as long as needed. However, that is not possible to do in a backwards-compatible way. Instead, a minimal set of fields is placed inside a shared pointer, namely:

- The `queue_mutex`
- A flag indicating whether the stream has stopped.

For convenience, the flag is encoded as a pointer, which holds either a pointer to the stream (if not stopped) or a null pointer (if stopped). Each completion handler holds a shared reference to this structure. When it wishes to access the stream, it should:

1. Lock the mutex.
2. Get the pointer back to the stream from the shared structure, aborting if it gets a null pointer.
3. Manipulate the stream.
4. Drop the mutex.

This prevents use-after-free errors because the stream cannot be destroyed without first stopping, and stopping locks the mutex. Hence, the stream cannot disappear asynchronously during step 3. Note that it can, however, stop during step 3 if the completion handler causes it to stop. Some protection is added for this: `stream::add_packet_handler::add_packet()` will not immediately stop the stream if a stop packet is received; instead, it will stop it when the `stream::add_packet_handler` is destroyed.

Using shared pointers in this way can add overhead because atomically incrementing and decrementing reference counts can be expensive, particularly if it causes cache line migrations between processor cores. To minimise reference count manipulation, the `reader` class encapsulates this workflow in its `bind_handler()` member function, which provides the facilities to move the shared pointer along a linear chain of completion handlers so that the reference count does not need to be adjusted.

Readers are only destroyed when the stream is destroyed. This ensures that the reader's destructor is called from a user's thread (which in Python bindings, will hold the GIL). To handle more immediate cleanup when a stream is stopped, readers may override `reader::stop()`.

9.2.4 Synchronisation in chunk stream groups

For chunk stream groups to achieve the goal of allowing multi-core scaling, it is necessary to minimise locking. The implementation achieves this by avoiding any packet- or heap-granularity locking, and performing locking only at chunk granularity. Chunks are assumed to be large enough that this minimises total overhead, although it should be noted that these locks are expected to be highly contended and there may be further work possible to reduce the overheads.

To avoid the need for heap-level locking, each member stream has its own sliding window with pointers to the chunks, so that heaps which fall inside an existing chunk can be serviced without locking. However, this causes a problem when flushing chunks from the group's window: a stream might still be writing to the chunk at the time. Additionally, it might

not be possible to allocate a new chunk until an old chunk is flushed e.g., if there is a fixed pool of chunks rather than dynamic allocation.

The group keeps its own copy of the head positions (oldest chunk) from the individual streams, protected by the group mutex rather than the stream mutexes. The group then maintains its head chunk position to match the oldest head position of any of the member streams. When the group wishes to evict a chunk, it simply needs to wait for all streams to make enough progress that the group's head moves past that chunk.

The wait is achieved using a condition variable that is notified whenever the head position increases. This allows the group mutex to be dropped while waiting, which prevents the deadlocks that might otherwise occur if the mutex was held while waiting and another stream was attemping to lock the group mutex to make forward progress.

In lossless eviction mode, this is all that is needed, although it is non-trivial to see that this won't deadlock with all the streams sitting in the wait loop waiting for other streams to make forward progress. That this cannot happen is due to the requirement that the stream's window cannot be larger than the group's. Consider the active call to `chunk_stream_group::get_chunk()` with the smallest chunk ID. That stream is guaranteed to have already readied any chunk due to be evicted from the group, and the same is true of any other stream that is waiting in `get_chunk()`, and so forward progress depends only on streams that are not blocked in `get_chunk()`.

In lossy eviction mode, we need to make sure that such streams make forward progress even if no new packets arrive on them. This is achieved by posting an asynchronous callback to all streams requesting them to flush out chunks that are now too old. The callback will never reach streams that have already stopped; we handle this at the time the stream stops, by treating it as having a head of `INT64_MAX`.

While lossless mode is normally allowed to block indefinitely, we do need to interrupt things in *`chunk_stream_group::stop()`*. This is handled similarly to lossy eviction mode, where all streams are requested to flush up to `INT64_MAX`.

9.2.5 Send rate limiting

The basic principle behind the rate limiting is that is that after sending a packet, one should sleep for some time before sending the next packet, to create gaps on the wire. However, there are a number of challenges:

1. Checking the time has some cost, and sleeping has quite a large cost. Sleeping after every packet can add so much cost that one can't keep up with the desired rate.
2. The OS can be late to wake up the process after a sleep. If not compensated for, this oversleeping time will reduce the achieved rate.
3. Naïvely catching up from oversleeping by transmitting as fast as possible can lead to a large burst of back-to-back packets that overwhelm the receiver.

The first point is addressed by sending several packets at a time without sleeping in between. Apart from reducing the number of sleeps, this also allows multiple packets to be batched together for transmission with APIs such as `sendmmsg()`. This is the *burst_size* parameter in *`StreamConfig`*.

The remaining points are handled by using two rates: the "standard" rate that the user requested, and a "catch-up" rate that is used when it is necessary to catch up after oversleeping, and which is specified indirectly via the *burst_rate_ratio* parameter in *`StreamConfig`*.

Note: While both parameters have "burst" in the name, they control two different bursting mechanisms: sending small amounts with no sleeping at all, and sending larger amounts at the burst rate to catch up on oversleeping.

The two rates are managed by keeping two lower bounds for sending the next burst. For the standard rate, the time is incremented after each burst according only to the size of the burst, without considering actual transmission times. For the burst rate, the time for the next burst is the time that the current burst was *actually* sent plus the size over the rate.

The above all assumes that the producer always has some data to send, but in some applications the sender may go dormant for some extended time. When it starts again, a naïve implementation might interpret this dormant period as oversleeping and switch to the burst rate to catch up. To avoid this, the rate mechanism handles this case specially by adjusting the standard rate lower bound such that no catching up is required.

State machine

The `writer` is a state machine which the following states:

- **New**: freshly constructed. Nothing happens in this state, because the associated stream has not yet been set.
- **Active**: The writer is either executing code or has made internal arrangements to be woken up (for example, it has asynchronously sent some packets and is waiting for the completion handler).
- **Sleeping**: The rate limiter is sleeping.
- **Empty**: All the queued heaps have been sent, and we are waiting for the user to provide more.

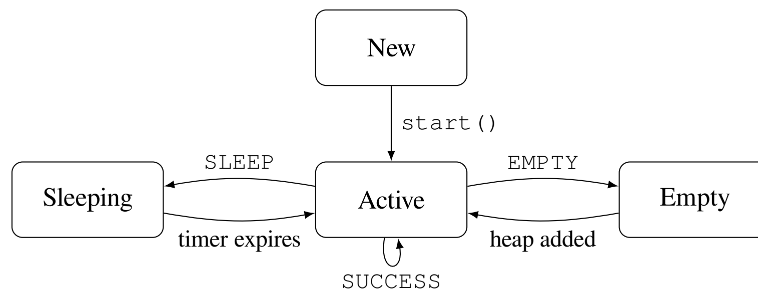

Fig. 1: State transitions

The transitions from **Active** are labelled by the return value from `spead2::send::writer::get_packet()`. Transitions back to **Active** are achieved by calling `spead2::send::writer::wakeup()`.

Time precision

Even though sleeping is not very precise, it has turned out to be necessary to do time arithmetic with very high (sub-nanosecond) precision. The reason is that standard rate lower bound will typically be incremented by the same amount for each burst, and hence any rounding error will be in the same direction each time. As an example, suppose the desired rate is 40 Gb/s, and each burst is 65536 bytes. Then the time between bursts should be 13107.2 ns. If arithmetic were done at nanosecond precision, that would round to 13107 ns each time, giving an actual rate of 40.0006 Gb/s. The higher the rate or the smaller the burst, the greater the relative error.

This is handled by representing absolute times as the sum of two parts: a `time_point` of the timer class (typically nanosecond resolution), and an additional correction in double precision (always between 0 and 1 units of `time_point`). When actually sleeping, only the first ("coarse") part is used, since that is all the precision that can be given to the timer. The correction term accumulates the rounding errors so that they do not get lost. Keeping the correction in the interval [0, 1) simplifies comparison of precise times.

9.2.6 Generic segmentation offload

Linux supports a mechanism called *generic segmentation offload* (GSO) to reduce packet overheads when transmitting UDP data through the kernel networking stack. A good overview can be found on Cloudflare's blog, but the basic idea is this:

1. Userspace concatenates multiple smaller packets into one mega-packet for submission to the kernel.
2. Most of the networking stack operates on the mega-packet.
3. As late as possible (and possibly on the NIC) the mega-packet is re-segmented into the original packets.

The re-segmentation uses a user-supplied parameter (socket option) indicating the size of the original packets. This imposes a limitation that the original packets were all the same size, except perhaps for the last one in the mega-packet.

The support for this in spead2 is dependent on the *sendmmsg(2)* support. While there is no fundamental reason GSO can't be used without *sendmmsg(2)*, supporting it would complicate the code significantly, and GSO is a much more recent feature so it is unlikely that this combination would ever be needed.

Run-time detection of support is unfortunately rather complicated. The simple part is that an older kernel will not support the socket option. If that occurs, we simply disable GSO for the stream. A more tricky problem is that actually sending the message may fail for several reasons:

- Fragmentation doesn't seem to be supported, so if the segment size is bigger than the MTU, it will fail.
- If hardware checksumming is disabled (or presumably if it is not supported), it will fail.

To cope with this complication, a state machine is used. It has four possible states:

- **active**: the socket option is set to a positive value
- **inactive**: the socket option is set to zero, but we may still transition to active
- **probe**: the last send in active state failed; the socket option is now set to zero and we're retrying
- **disabled**: the socket option is set to zero, and we will never try to set it again.

If send fails while in state **active**, we switch to state **probe** and try again (without GSO). If that succeeds, we conclude that GSO is non-functional for this stream and permanently go to **disabled**. If that also fails, we conclude that the problem was unrelated to GSO and return to **inactive**.

9.2.7 Linking to ibverbs

The Python wheels for spead2 support *ibverbs*. But what happens if the ibverbs library isn't installed on the user's machine? While for pcap we simply bundle a copy of the library into the wheel, this is problematic for ibverbs: it uses configuration files in `/etc/libibverbs.d` to configure drivers, which are themselves contained in other shared libraries. At best, one would end up with a mix of code from the wheel and from the operating system, which could easily lead to compatibility problems.

Instead, the ibverbs libraries (`libibverbs.so`, `librdmacm.so` and `libmlx5.so`) are loaded dynamically at runtime with dlopen(3). If the library is not found, the corresponding functionality in spead2 is disabled.

Unfortunately, dynamic loading is not easy to use: dlsym(3) returns a raw pointer with no type information. One needs to check for errors and cast the pointer to the proper type before it can be invoked. This leads to a lot of boilerplate code, which in spead2 is generated at build time by `gen/gen_loader.py` for each of the libraries.

The process starts with the signatures for the functions that are to be written, which are parsed using pycparser. The actual header files for these functions are not suitable for pycparser (it only handles standard C99), so the script has embedded declarations for all the functions we wish to wrap. It then emits a header file and a source file, both generated from templates using jinja2. It generates a private initialisation function that opens the library (if possible) and extracts the symbols from it. It also generates the following for each function:

- A declaration of the function pointer. It has the same name as the original function, but is in the `spead2` namespace to prevent symbol conflicts.
- An implementation that calls the initialisation function (using `std::call_once()`, to avoid initialising multiple times), then calls through the function pointer. The function pointer is defined to initially point to this implementation, so that the first use of the function pointer does the initialisation.
- A "stub" implementation. If there was a problem loading the library, the function pointers are all changed to point to the stub implementations, which when called will re-throw the exception that occurred during initialisation.

Additionally, some functions are optional: they were added in later versions of the wrapped library, and spead2 has fallback code that can be used if the function is not available. For those, the generator additionally creates:

- A "missing" implementation, which throws a `std::system_error` with error code `EOPNOTSUPP`. If the function is not found during initialisation, the function pointer is set to point at this implementation.
- A (public) function which has the same name but prefixed with `has_`, which returns a boolean to indicate whether the function is present.

Changing a function pointer during initialisation ensures that only the first call incurs any overheads from the wrapping; after that, calls are made directly to the target function.

CHAPTER TEN

CHANGELOG

Development version

- Speed up receiving UDP with the Linux kernel network stack by using generic receive offload (GRO).
- Update Boost version in wheels to 1.85.

4.3.1

- Switch from netifaces to netifaces2 for testing.
- Update coverallsapp/github-action to v2.2.3.
- Fix the type annotation for `spead2.recv.StreamConfig` to allow *explicit_start* to be passed as a constructor argument.

4.3.0

- Add ability to override the transmission rate for individual heaps.
- Fix missing type annotation on the *substream_index* argument to `send_heap()`.

4.2.0

- Significantly speed up transmission when using the Linux kernel networking stack, by using generic segmentation offload.
- Speed up transmission for small packets (20% in some cases).
- Make `async_send_heap()` and `async_send_heaps()` accept completion tokens. This allows for code like `stream.async_send_heap(heap, boost::asio::use_future);`
- Add C++ support to iterate over `ring_stream` and `ringbuffer` using a range-based `for` loop.
- Add support for the *prefer_huge* argument to the `MmapAllocator` constructor in Python.
- Make the *allocator* argument to `MemoryPool` optional in Python.
- Allow indexing a `HeapReferenceList` with a slice to create a new `HeapReferenceList` with a subset of the heaps.
- Add `ringbuffer` to the C++ documentation.
- Eliminate use of the deprecated `boost::asio::null_buffers`.
- Update Boost version in wheels to 1.84.

- Update rdma-core version in manylinux wheels to 49.0.
- Make various updates to the Github Actions infrastructure.

4.1.1

- Add AVX and AVX-512 implementations for non-temporal memory copy. While this is transparent to the API, the Meson option names have changed to allow specific instruction sets to be disabled if necessary.

4.1.0

- Introduce *explicit start* for receive streams.

4.0.2

Note that an oversight lead to some of the changes between 4.0.0b1 and 4.0.0 being omitted from 4.0.1. They are restored in 4.0.2.

- Fix type annotations for `spead2.send.UdpStream` and `spead2.send.asyncio.UdpStream`.
- Add more documentation for developers.
- Remove an old `Makefile.am` that should have been removed in 4.0.0.
- Remove mocking of spead2 in readthedocs build.
- Change .. *code::* to .. *code-block::* in documentation.
- Simplify the implementation of `thread_pool_wrapper` and `buffer_reader` in Python binding code.
- Directly use pointer-to-member-functions in Python binding code.
- Test against numpy 1.26.0 (instead of 1.26.0rc1) on Python 3.12.

4.0.1

- Restore dependency on numpy, which was accidentally removed in 4.0.0.
- Change the `test_numba` extra to `test-numba` to normalise it in accordance with **PEP 685**.

4.0.0

This release makes major changes to the build system, and removes deprecated features. See *Migrating to version 4* for more detailed information about upgrading and the deprecation removals.

Most of the changes are listed under 4.0.0b1 below. Since then, the following changes have been made:

- Improve detection of gdrcopy by using the CUDA compiler to compile the test code.
- Remove ninja from `build-system.requires`. If you have ninja installed on the system, that will be used for the Python install rather than downloading it.
- Make miscellaneous improvements to the build system.
- Remove an unused file (`.ci/ccache-path.sh`).
- Work around a pytest bug to prevent tests running out of file descriptors (particularly on MacOS, which has a lower default limit).

- Add wheels for MacOS (Intel and Apple Silicon).
- Document that Meson must be at least 1.2.
- Make source paths in `.debug` files more usable (relative to 4.0.0b1).
- Remove `.pyi` file entries for the functionality removed in 4.0.

4.0.0b1

This release makes major changes to the build system, and removes deprecated features. See *Migrating to version 4* for more detailed information about upgrading and the deprecation removals.

- Replace the build system with Meson.
- Update the C++ code to use C++17.
- No longer link against boost_system, and require Boost 1.69+.
- Remove generated code from release tarballs; some Python packages are now required at build time to build the C++ bindings.
- Fix an uninitialised variable that could cause a segmentation fault when using TCP to send and the initial connection failed.
- Fix a large number of compiler warnings that showed up after switching build systems (mainly related to unused function parameters and signed/unsigned comparisons).
- Fix the debug logging option so logging from inline functions in headers will also do debug logging without the user needing to make preprocessor defines.
- Fix **`spead2_bench.py`** so that it functions when ibverbs support is not compiled in.
- Remove the need for boost_program_options to be installed to be able to install the Python bindings from source.
- Produce binary wheels for aarch64.
- Produce wheels for Python 3.12.
- Make numba and scipy optional for running tests (some tests will be skipped if they are not present).
- Update the libpcap embedded in the wheels to 1.10.4.
- Update the Boost version used to build wheels to 1.83.
- Update the rdma-core version used to build wheels to 47.0.
- Update the pybind11 build dependency to 2.11.1.
- Replace flake8 with ruff for linting.
- Remove the `spead2/common_bind.h` header, which was unused.
- Remove the `SPEAD2_DEPRECATED` macro.
- Remove build-time dependencies from `requirements.txt`.
- Update the `.pyi` files to use more modern syntax e.g., **PEP 585**, **PEP 604**, **PEP 613**.
- Replace references to nv_peer_mem with nvidia-peermem.
- Increase TTL of **`gpudirect_example`** to 4.

3.13.0

- Reformat the Python codebase using black and isort.
- Add pre-commit configuration.
- On i386, check for SSE2 support at runtime rather than configure time.
- Free readers only when the stream is destroyed. This fixes a bug that caused the Python API to be accessed without the GIL when using `add_buffer_reader()`.
- Improve unit tests by explicitly closing TCP sockets, to avoid `ResourceWarning` when testing with `python -X dev`.
- Remove `wheel` from `build-system.requires`.

3.12.0

- Add support for *Chunking stream groups* to assemble chunks in parallel.
- Simplify the way receive streams shut down. Users should not notice any change, but custom reader implementations will need to be updated.
- Update `test_async_flush()` and `test_async_flush_fail()` to keep handles to async tasks, to prevent them being garbage collected too early.
- Fix a bug where copying a `spead2::recv::stream_config` would not deep copy the names of custom statistics, and so any statistics added to the copy would also affect the original, and there were also potential race conditions if a stream config was modified while holding stream statistics.
- Fix a bug (caused by the bug above) where passing a `spead2::recv::stream_config` to construct a `spead2::recv::chunk_stream` would modify the config. Passing the same config to construct two chunk streams would fail with an error.
- Fix the type annotation for the `ChunkRingStream` constructor: the parameter name for *chunk_stream_config* was incorrect.
- Fix universal binary builds on MacOS (this was preventing Python 3.11 builds from succeeding).
- Fix **`spead2_bench.py`**, which has failed to run at all for some time (possibly since 3.0).
- Avoid including Boost dynamic symbols in the Python module (helps reduce binary size).
- Strip static symbols out of the Python wheels (reduces size).
- Build Python wheels with link-time optimisation (small performance improvement).
- Python 3.8 is now the minimum version.

3.11.1

- Fix a packaging issue that meant automake and similar tools were required to compile (since 3.10).

3.11.0

- The chunking receiver is no longer experimental.
- The place callback for the chunking receiver can now provide extra data to be written to the chunk.

3.10.0

- Support pcap dumps that use the SLL format.
- Support a user-defined filter in the pcap file reader.
- Add experimental support for building a shared library.
- Assorted documentation updates
  - The SPEAD specification is now stored in the repository (the upstream link is broken).
  - Build PDFs on readthedocs.
  - Update the tuning documentation.

3.9.1

- Fix an `asyncio.InvalidStateError` that occurs when the future returned by `async_send_heap()` or `async_send_heaps()` is cancelled before it completes.

3.9.0

- Added `substreams` to `spead2.recv.StreamConfig` to improve handling of interleaved heaps from multiple senders.
- Add libdivide to the dependencies.

3.8.0

- Drop support for Python 3.6, which has reached end-of-life.
- Test against Python 3.10 in Github Actions.
- Improve the accuracy of the rate limiter. Previously it could send slightly too fast due to rounding sleep times to whole numbers of nanoseconds.
- Eliminate dependence on distutils, which is deprecated in Python 3.10 (#175).

3.7.0

- Add `spead2.send.GroupMode.SERIAL`.
- Add `spead2.send.HeapReferenceList`.
- Speed up C++ unit tests.
- Fix some spurious output in the statistics report from **spead2_recv.py** (introduced in 3.5.0).
- Fix the help message from **spead2_net_raw** to have the right name for the program.
- Update to latest version of pybind11.

3.6.0

- Allow a ringbuffer to be stopped only once the last producer has indicated completion, rather than the first.
- Change `ChunkRingStream` so that stops received from the network only shut down a shared ringbuffer once all the streams have stopped. A user call to `stop` will still stop the ringbuffer immediately.
- `ChunkRingbuffer.stop()` now returns a boolean to indicate whether this is the first time the ringbuffer was stopped.

3.5.0

- Add support for *Custom statistics*.
- Change the allocate and ready callbacks on `spead2::recv::chunk_stream` to take a pointer to the batch statistics. This is a **backwards-incompatible change** (keep in mind that chunking receive is still experimental). Code that uses `spead2::recv::chunk_ring_stream` is unaffected.
- Change the design of deleters for `spead2::memory_allocator`. Code that calls `get_user` or `get_deleter` on a pointer allocated by spead2 may now get a `nullptr` back. Code that uses a custom memory allocator and that calls these functions on pointers allocated by that allocator should continue to work.
- Allow a ready callback to be used together with `spead2::recv::chunk_ring_stream`, to finish preparation of a chunk before it pushed to the ringbuffer.
- In Python, avoid copying immediate items when given as 0-d arrays with dtype `>u8`. This makes it practical to pre-define heaps and later update their values rather than creating new heap objects.
- Make `spead2.send.Stream`, `spead2.send.SyncStream` and `spead2.send.asyncio.AsyncStream` available for type annotations.
- Fix an occasional segfault when stopping a `spead2.recv.ChunkRingStream`.

3.4.0

- Add *Chunking receiver*.
- Add missing `spead2.recv.Stream.add_udp_pcap_file_reader()` to .pyi file.
- Add `spead2.InprocQueue.add_packet()`.
- Prevent conversions from `None` to `spead2.ThreadPool`.

3.3.2

- `spead2::recv::mem_reader` now stops the stream gracefully, allowing incomplete heaps to be flushed.

3.3.1

- Convert **spead2_net_raw** to a C++ file so that it gets the same compiler flags as everything else.
- Migrate from Travis CI to Github Actions.
- Fix some warnings generated by Clang.
- Fix some test failures with PyPy.

3.3.0

- Add *spead2_net_raw* tool.
- Eliminate some compiler warnings about unused parameters.
- Update build process to use pypa-build and setuptools_scm.
- Update to pybind11 2.6.2.

3.2.2

- Use `python3` instead of `python` to invoke Python (so that it works even on systems where `python` is absent or is Python 2).
- Work around a bug that prevented compilation on Boost 1.76.

3.2.1

- Update type annotations to use `numpy.typing.DTypeLike` for dtype arguments, to prevent false warnings from mypy.

3.2.0

- Add *`spead2::recv::heap::get_payload()`* to allow the payload pointer to be retrieved from a complete heap.
- Make the ibverbs sender compatible with PeerDirect.
- Add examples programs showing integration with gdrcopy and PeerDirect.
- Always use SFENCE at end of `memcpy_nontemporal()` so that it is appropriate for use with gdrcopy.
- Fix a memory leak when receiving with ibverbs.

3.1.3

- Fix installation of header files: some newer headers were not being installed, breaking builds for C++ projects.

3.1.2

- Fix a use-after-free bug that could cause a crash when freeing a send stream.
- Improve send performance by eliminating a memory allocation from packet generation.

3.1.1

- Set `IBV_ACCESS_RELAXED_ORDERING` flag on ibverbs memory regions. This reduces packet loss in some circumstances (observed on Epyc 2 system with lots of memory traffic).

3.1.0

- Add `send_heaps()` and `async_send_heaps()` to send groups of heaps with interleaved packets.
- Upgrade to pybind11 2.6.0, which contains a workaround for a bug in CPython 3.9.0.

3.0.1

- Bring the type stubs up to date.
- Fix a typo in the documentation.

3.0.0

Version 3.0 contains a number of breaking API changes. For information on updating your existing code, refer to *Migrating to version 3*.

The *ibverbs* acceleration has been substantially modified to use a newer version of rdma-core. It will no longer compile against versions of MLNX-OFED prior to 5.0. Compiled code (such as Python wheels) will still run against old versions of MLNX-OFED, but extension features such as multi-packet receive queues and packet timestamps will not work, and nor will **mcdump**. It is recommended that if you are using ibverbs acceleration with older MLNX-OFED drivers that you stick with spead2 2.x until you're able to upgrade the drivers and spead2 simultaneously.

- Support multiple "substreams" in a send stream (see *Substreams*).
- Reduce overhead for dealing with incomplete heaps.
- Allow ibverbs senders to register memory regions for zero-copy transmission.
- Add C++ preprocessor defines for the version number.
- Use IP/UDP checksum offloading for sending with ibverbs (improves performance and also adds UDP checksum which is otherwise omitted).
- Add wheels for Python 3.9.
- Drop support for Python 3.5, which is end-of-life.
- Change code examples to use standard SPEAD rather than PySPEAD bug compatibility.
- Change `spead2::send::streambuf_stream` so that when the streambuf only partially writes a packet, the partial byte count is included in the count returned to the callback.
- `spead2::send::stream::flush()` now only blocks until the previously enqueued heaps are completed. Another thread that keeps adding heaps would previously have prevented it from returning.
- Partially rewrite the sending infrastructure, resulting in performance improvements, in some cases of over 10%.

- Setting a buffer size of 0 for a *UdpIbvStream* now uses the default buffer size, instead of a 1-packet buffer.
- Fix **spead2_bench.py** ignoring the `--send-affinity` option.
- Add `--verify` option to **spead2_send** and **spead2_recv** to aid in testing the code. To support this, **spead2_send** was modified so that each in-flight heap uses different memory, which may reduce performance (due to less cache re-use) even when the option is not given.
- Miscellaneous performance improvements.
- Support hardware send rate limiting when using ibverbs (disabled by default).
- Discover libibverbs and pcap using pkg-config where possible.
- Make **configure** print out the configuration that will be compiled.
- Update the Python wheels to use manylinux2014. This uses a newer compiler (potentially giving better performance) and supports `sendmmsg()`.
- A number of deprecated functions have been removed.
- Avoid ibverbs code creating a send queue for receiver or vice versa.
- Rename `slave` option to **spead2_bench** to `agent`.

Compared to 3.0.0b2 there is a critical bug fix for a race condition in the send code.

3.0.0b2

Version 3.0 contains a number of breaking API changes. For information on updating your existing code, refer to *Migrating to version 3*.

Other changes:

- Support multiple "substreams" in a send stream (see *Substreams*).
- Reduce overhead for dealing with incomplete heaps.
- Allow ibverbs senders to register memory regions for zero-copy transmission.
- Add C++ preprocessor defines for the version number.
- Use IP/UDP checksum offloading for sending with ibverbs (improves performance and also adds UDP checksum which is otherwise omitted).
- Drop support for Python 3.5, which is end-of-life.
- Change code examples to use standard SPEAD rather than PySPEAD bug compatibility.
- Change *spead2::send::streambuf_stream* so that when the streambuf only partially writes a packet, the partial byte count is included in the count returned to the callback.
- *spead2::send::stream::flush()* now only blocks until the previously enqueued heaps are completed. Another thread that keeps adding heaps would previously have prevented it from returning.
- Partially rewrite the sending infrastructure, resulting in performance improvements, in some cases of over 10%.
- Setting a buffer size of 0 for a *UdpIbvStream* now uses the default buffer size, instead of a 1-packet buffer.
- Fix **spead2_bench.py** ignoring the `--send-affinity` option.
- The hardware rate limiting introduced in 3.0.0b1 is now disabled by default, as it proved to be significantly less accurate than the software rate limiter in some cases. The interface has also been changed from a boolean to an enum (with the default being `AUTO`) so that it can later be re-enabled under circumstances where it is known to work well, while still allowing it to be explicitly enabled or disabled.

- Add `--verify` option to **spead2_send** and **spead2_recv** to aid in testing the code. To support this, **spead2_send** was modified so that each in-flight heap uses different memory, which may reduce performance (due to less cache re-use) even when the option is not given.
- Miscellaneous performance improvements.

Additionally, refer to the changes for 3.0.0b1 below.

3.0.0b1

The *ibverbs* acceleration has been substantially modified to use a newer version of rdma-core. It will no longer compile against versions of MLNX-OFED prior to 5.0. Compiled code (such as Python wheels) will still run against old versions of MLNX-OFED, but extension features such as multi-packet receive queues and packet timestamps will not work. It is recommended that if you are using ibverbs acceleration with older MLNX-OFED drivers that you stick with spead2 2.x until you're able to upgrade the drivers and spead2 simultaneously.

Other changes:

- Support hardware send rate limiting when using ibverbs.
- Discover libibverbs and pcap using pkg-config where possible.
- Make **configure** print out the configuration that will be compiled.
- Update the Python wheels to use manylinux2014. This uses a newer compiler (potentially giving better performance) and supports `sendmmsg()`.
- Add wheels for Python 3.9.
- A number of deprecated functions have been removed.
- Avoid ibverbs code creating a send queue for receiver or vice versa.
- Rename `slave` option to **spead2_bench** to `agent`.

2.1.2

- Make verbs acceleration work when run against MLNX OFED 5.x, including with Python wheels. Note that it will not use multi-packet receive queues, so receive performance may still be better on MLNX OFED 4.9.

2.1.1

- Update pybind to 2.5.0.
- Fix compilation against latest rdma-core.
- Some documentation cleanup.

2.1.0

- Support unicast receive with ibverbs acceleration (including in **mcdump**).
- Fix **spead2_recv** listening only on loopback when given just a port number.
- Support unicast addresses in a few APIs that previously only accepted multicast addresses; in most cases the unicast address must match the interface address.
- Add missing `<map>` include to `<spead2/recv_heap.h>`.
- Show the values of immediate items in **spead2_recv**.
- Fix occasional crash when using thread pool with more than one thread together with ibverbs.
- Fix bug in mcdump causing it to hang if the arguments couldn't be parsed (only happened when capturing to file).
- Fix **spead2_recv** reporting statistics that may miss out the last batch of packets.

2.0.2

- Log warnings on some internal errors (that hopefully never happen).
- Include wheels for Python 3.8.
- Build debug symbols for binary wheels (in a separate tarball on Github).

2.0.1

- Fix race condition in TCP receiver (#78).
- Update vendored pybind11 to 2.4.2.

2.0.0

- Drop support for Python 2.
- Drop support for Python 3.4.
- Drop support for trollius.
- Drop support for netmap.
- Avoid creating some cyclic references. These were not memory leaks, but prevented CPython from freeing objects as soon as it might have.
- Update vendored pybind11 to 2.4.1.

1.14.0

- Add *new_order* argument to `spead2.ItemGroup.update()`.
- Improved unit tests.

1.13.1

- Raise `ValueError` on a dtype that has zero itemsize (#37).
- Change exception when dtype has embedded objects from `TypeError` to `ValueError` for consistency
- Remove duplicated socket handle in UDP receiver (#67).
- Make *max_poll* argument to `spead2.send.UdpIbvStream` actually have an effect (#55).
- Correctly report EOF errors in `spead2::send::streambuf_stream`.
- Wrap implicitly computed heap cnts to the number of available bits (#3). Previously behaviour was undefined.
- Some header files were not installed by `make install` (#72).

1.13.0

- Significant performance improvements to send code (in some cases an order of magnitude improvement).
- Add `--max-heap` option to **spead2_send** and **spead2_send.py** to control the depth of the send queue.
- Change the meaning of the `--heaps` option in **spead2_bench** and **spead2_bench.py**: it now also controls the depth of the sending queue.
- Fix a bug in send rate limiting that could allow the target rate to be exceeded under some conditions.
- Remove `--threads` option from C++ **spead2_send**, as the new optimised implementation isn't thread-safe.
- Disable the `test_numpy_large` test on macOS, which was causing frequent failures on TravisCI due to dropped packets.

1.12.0

- Provide manylinux2010 wheels.
- Dynamically link to libibverbs and librdmacm on demand. This allows binaries (particularly wheels) to support verbs acceleration but still work on systems without these libraries installed.
- Support for Boost 1.70. Unfortunately Boost 1.70 removes the ability to query the io_service from a socket, so constructors that take a socket but no io_service are omitted when compiling with Boost 1.70 or newer.
- Fix some compiler warnings from GCC 8.

1.11.4

- Rework the locking internals of `spead2::recv::stream` so that a full ringbuffer doesn't block new readers from being added. This changes the interfaces between `spead2::recv::reader` and `spead2::recv::stream_base`, but since users generally don't deal with that interface the major version hasn't been incremented.
- Fix a spurious log message if an in-process receiver is manually stopped.
- Fix an intermittent unit test failure due to timing.

1.11.3

- Undo the optimisation of using a single flow steering rule to cover multiple multicast groups (see #11).

1.11.2

- Fix `-c` option to **mcdump**.
- Fix a missing `#include` that could be exposed by including headers in a particular order.
- Make `spead2::recv::heap`'s move constructor and move assignment operator `noexcept`.
- Add a *long_description* to the Python metadata.

1.11.1

- Update type stubs for new features in 1.11.0.

1.11.0

- Add `spead2.recv.Stream.allow_unsized_heaps` to support rejecting packets without a heap length.
- Add extended custom memcpy support (C++ only) for scattering data from packets.

1.10.1

- Use ibverbs multi-packet receive queues automatically when available (supported by mlx5 driver).
- Automatically reduce buffer size for verbs receiver to match hardware limits (fixed #64).
- Gracefully handle Ctrl-C in **spead2_recv** and print statistics.
- Add typing stub files to assist checking with Mypy.
- Give a name to the argument of `spead2.recv.Stream.add_inproc_reader()`.
- Fix Python binding for one of the UDP reader overloads that takes an existing socket. This was a deprecated overload.
- Add a unit test for ibverbs support. It's not run by default because it needs specific hardware.

1.10.0

- Accelerate per-packet processing, particularly when *max_heaps* is large.
- Accelerate per-heap processing, particularly for heaps with few items.
- Add a fast path for single-packet heaps.
- Improve performance of the pcap reader by working on batches of packets.
- Provide access to ringbuffer size and capacity for diagnostics.
- Add extra fields to `spead2.recv.StreamStats`.
- Add support for pcap files to the C++ version of **spead2_recv**.
- Update the vendored pybind11 to 2.2.4 (fixes some warnings on Python 3.7).

- Deprecate netmap support in documentation.

1.9.2

- autotools are no longer required to install the C++ build (when installing from a release tarball).

1.9.1

- Make `spead2.recv.asyncio.Stream.get()` always yield to the event loop even if there is a heap ready.
- Avoid `spead2.recv.asyncio.Stream.get()` holding onto a reference to the heap (via a future) for longer than necessary.

1.9.0

- Add support for TCP/IP (contributed by Rodrigo Tobar).
- Changed command-line options for **spead2_send/spead2_recv**: `--ibv` and `--netmap` are now boolean flags, and the interface address is set with `--bind`.
- Added option to specify interface address for `spead2::send::udp_stream` even when not using the multicast constructors.
- Constructors that take an existing socket now expect the user to set all socket options. The old versions that take a socket buffer size are deprecated. Note that the behaviour of `spead2::send::udp_stream` with a socket has **changed**: if no buffer size is given, it is left at the OS default, rather than applying the spead2 default.
- Fix a bug causing undefined behaviour if a send class is destroyed while there is still data in flight.

Version 1.8.0

- Add *In-process transport*
- Fix unit testing on Python 3.7
- Add `spead2::send::heap::get_item()`
- Support asynchronous iterator protocol for `spead2.recv.asyncio.Stream` (in Python 3.5+).

Version 1.7.2

- Add progress reports to mcdump
- Add ability to pass `-` as filename to mcdump to skip file writing.
- Add `--count` option to mcdump

Version 1.7.1

There are no code changes, but this release fixes a packaging error in 1.7.0 that prevented the asyncio integration from being included.

Version 1.7.0

- Support for pcap files. Files passed to **spead2_recv.py** are now assumed to be pcap files, rather than raw concatenated packets.
- Only log warnings about the ringbuffer being full if at least one stream reader is lossy (indicated by a new virtual member function in `spead2::recv::Reader`).

Version 1.6.0

- Change **spead2_send.py** and **spead2_send** to interpret the `--rate` option as Gb/s and not Gib/s.
- Change send rate limiting to bound the rate at which we catch up if we fall behind. This is controlled by a new attribute of *StreamConfig*.
- Add report at end of **spead2_send.py** and **spead2_send** on the actual number of bytes sent and achieved rate.
- Fix a race condition where the stream statistics might only be updated after the stream ended (which lead to unit test failures in some cases).

Version 1.5.2

- Report statistics when **spead2_recv.py** is stopped by SIGINT.
- Add –ttl option to **spead2_send.py** and **spead2_send**.

Version 1.5.1

- Explicitly set UDP checksum to 0 in IBV sender, instead of leaving arbitrary values.
- Improved documentation of asyncio support.

Version 1.5.0

- Support for asyncio in Python 3. For each trollius module there is now an equivalent asyncio module. The installed utilities use asyncio on Python 3.4+.
- Add `spead2.recv.Stream.stop_on_stop_item` to allow a stream to keep receiving after a stop item is received.
- Switch shutdown code to use atexit instead of a capsule destructor, to support PyPy.
- Test PyPy support with Travis.

Version 1.4.0

- Remove `--bind` option to **spead2_recv.py** and **spead2_recv**. Instead, use *host*:*port* as the source. This allows subscribing to multiple multicast groups.
- Improved access to information about incomplete heaps (`spead2.recv.IncompleteHeap` type).
- Add `MemoryPool.warn_on_empty` control.
- Add warning when a stream ringbuffer is full.
- Add statistics to streams.
- Fix spead2_send.py to send a stop heap when using `--heaps`. It was acccidentally broken in 1.2.0.
- Add support for packet timestamping in mcdump.
- Return the previous logging function from `spead2::set_log_function()`.
- Make Python logging from C++ code asynchronous, to avoid blocking the thread pool on the GIL.
- Upgrade to pybind11 2.2.1 internally.
- Some fixes for PyPy support.

Version 1.3.2

- Fix segfault in shutdown for `spead2_recv.py` (fixes #56).
- Fix for `TypeError` in Python 3.6 when reading fields that aren't aligned to byte boundaries.
- Include binary wheels in releases.

Version 1.3.1

- Fix multi-endpoint form of `spead2.recv.Stream.add_udp_ibv_reader()`.

Version 1.3.0

- Rewrite the Python wrapping using pybind11. This should not cause any compatibility problems, unless you're using the `spead2/py_*.h` headers.
- Allow passing `std::shared_ptr` to constructors that take a thread pool, with the constructed object holding a reference.
- Prevent constructing a `spead2.recv.Stream` with `max_heaps=0` (fixes #54).

Version 1.2.2

- Fix rate limiting causing longer sleeps than necessary (fixes #53).

Version 1.2.1

- Disable LTO by default and require the user to opt in, because even if the compiler supports it, linking can still fail (fixes #51).

Version 1.2.0

- Support multiple endpoints for one `udp_ibv_reader` (fixes #48).
- Fix compilation on OS X 10.9 (fixes #49)
- Fix `spead2::ringbuffer<T>::emplace()` and `spead2::ringbuffer<T>::try_emplace()`
- Improved error messages when passing invalid arguments to mcdump

Version 1.1.2

- Only log descriptor replacement if it actually replaces an existing name or ID (regression in 1.1.1).
- Fix build on ARM where compiling against asio requires linking against pthread.
- Updated and expanded performance tuning guide.

Version 1.1.1

- Report the item name in exception for “too few elements for shape” errors
- Overhaul of rules for handling item descriptors that change the name or ID of an item. This prevents stale items from hanging around when the sender changes the name of an item but keeps the same ID, which can cause unrelated errors on the receiver if the shape also changes.

Version 1.1.0

- Allow heap cnt to be set explicitly by sender, and the automatic heap cnt sequence to be specified as a start value and step.

Version 1.0.1

- Fix exceptions to include more information about the source of the failure
- Add *mcdump* tool

Version 1.0.0

- The C++ API installation has been changed to use autoconf and automake. As a result, it is possible to run `make install` and get the static library, headers, and tools installed.
- The directory structure has changed. The `spead2_*` tools are now installed, example code is now in the `examples` directory, and the headers have moved to `include/spead2`.
- Add support for sending data using libibverbs API (previously only supported for receiving)
- Fix async_send_heap (in Python) to return a future instead of being a coroutine: this fixes a problem with undefined ordering in the trollius example.

- Made sending streams polymorphic, with abstract base class `spead2::send::stream`, to simplify writing generic code that can operate on any type of stream. This will **break** code that depended on the old template class of the same name, which has been renamed to `spead2::send::stream_impl`.
- Add `--memcpy-nt` to **spead2_recv.py** and **spead2_bench.py**
- Multicast support in **spead2_bench.py** and **spead2_bench**
- Changes to the algorithm for **spead2_bench.py** and **spead2_bench**: it now starts by computing the maximum send speed, and then either reporting that this is the limiting factor, or using it to start the binary search for the receive speed. It is also stricter about lost heaps.
- Some internal refactoring of code for dealing with raw packets, so that it is shared between the netmap and ibv readers.
- Report function name that failed in semaphore system_error exceptions.
- Make the unit tests pass on OS X (now tested on travis-ci.org)

Version 0.10.4

- Refactor some of the Boost.Python glue code to make it possible to reuse parts of it in writing new Python extensions that use the C++ spead2 API.

Version 0.10.3

- Suppress "operation aborted" warnings from UDP reader when using the API to stop a stream (introduced in 0.10.0).
- Improved elimination of duplicate item pointers, removing them as they're received rather than when freezing a live heap (fixes #46).
- Use hex for reporting item IDs in log messages
- Fix reading from closed file descriptor after stream.stop() (fixes #42)
- Fix segmentation fault when using ibverbs but trying to bind to a non-RDMA device network interface (fixes #45)

Version 0.10.2

- Fix a performance problem when a heap contains many packets and every packet contains item pointers. The performance was quadratic instead of linear.

Version 0.10.1

- Fixed a bug in registering *add_udp_ibv_reader* in Python, which broke **spead2_recv.py**, and possibly any other code using this API.
- Fixed **spead2_recv.py** ignoring `--ibv-max-poll` option

Version 0.10.0

- Added support for libibverbs for improved performance in both *Python* and *C++*.
- Avoid per-packet shared_ptr reference counting, accidentally introduced in 0.9.0, which caused a small performance regression. This is unfortunately a **breaking** change to the interface for implementing custom memory allocators.

Version 0.9.1

- Fix using a `MemoryPool` with a thread pool and low water mark (regression in 0.9.0).

Version 0.9.0

- Add support for custom memory allocators.

Version 0.8.2

- Ensure correct operation when *loop=None* is passed explicitly to trollius stream constructors, for consistency with functions that have it as a keyword parameter.

Version 0.8.1

- Suppress `recvmmsg: resource temporarily unavailable` warnings (fixes #43)

Version 0.8.0

- Extend `MemoryPool` to allow a background thread to replenish the pool when it gets low.
- Extend `ThreadPool` to allow the user to pin the threads to specific CPU cores (on glibc).

Version 0.7.1

- Fix ring_stream destructor to not deadlock (fixes #41)

Version 0.7.0

- Change handling of incomplete heaps (fixes #39). Previously, incomplete heaps were only abandoned once there were more than *max_heaps* of them. Now, they are abandoned once *max_heaps* more heaps are seen, even if those heaps were complete. This causes the warnings for incomplete heaps to appear closer to the time they arrived, and also has some extremely small performance advantages due to changes in the implementation.
- **backwards-incompatible change**: remove `set_max_heaps()`. It was not previously documented, so hopefully is not being used. It could not be efficiently supported with the design changes above.
- Add `spead2.recv.Stream.set_memcpy()` to control non-temporal caching hints.
- Fix C++ version of spead2_bench to actually use the memory pool
- Reduce memory usage in spead2_bench (C++ version)

Version 0.6.3

- Partially fix #40: `set_max_heaps()` and `set_memory_pool()` will no longer deadlock if called on a stream that has already had a reader added and is receiving data.

Version 0.6.2

- Add a fast path for integer items that exactly fit in an immediate.
- Optimise Python code by replacing np.product with a pure Python implementation.

Version 0.6.1

- Filter out duplicate items from a heap. It is undefined which of a set of duplicates will be retained (it was already undefined for `spead2.ItemGroup`).

Version 0.6.0

- Changed item versioning on receive to increment version number on each update rather that setting to heap id. This is more robust to using a single item or item group with multiple streams, and most closely matches the send path.
- Made the protocol enums from the C++ library available in the Python library as well.
- Added functions to create stream start items (send) and detect them (recv).

Version 0.5.0

- Added friendlier support for multicast. When a multicast address is passed to `add_udp_reader()`, the socket will automatically join the multicast group and set `SO_REUSEADDR` so that multiple sockets can consume from the same stream. There are also new constructors and methods to give explicit control over the TTL (send) and interface (send and receive), including support for IPv6.

Version 0.4.7

- Added in-memory mode to the C++ version of spead2_bench, to measure the packet handling speed independently of the lossy networking code
- Optimization to duplicate packet checks. This makes a substantial performance improvement when using small (e.g. 512 byte) packets and large heaps.

Version 0.4.6

- Fix a data corruption (use-after-free) bug on send side when data is being sent faster than the socket can handle it.

Version 0.4.5

- Fix bug causing some log messages to be remapped to DEBUG level

Version 0.4.4

- Increase log level for packet rejection from DEBUG to INFO
- Some minor optimisations

Version 0.4.3

- Handle heaps that have out-of-range item offsets without crashing (#32)
- Fix handling of heaps without heap length headers
- `spead2.send.UdpStream.send_heap()` now correctly raises `IOError` if the heap is rejected due to being full, or if there was an OS-level error in sending the heap.
- Fix `spead2.send.trollius.UdpStream.async_send_heap()` for the case where the last sent heap failed.
- Use *eventfd(2)* for semaphores on Linux, which makes a very small improvement in ringbuffer performance.
- Prevent messages about descriptor replacements for descriptor reissues with no change.
- Fix a use-after-free bug (affecting Python only).
- Throw `OverflowError` on out-of-range UDP port number, instead of wrapping.

Version 0.4.2

- Fix compilation on systems without glibc
- Fix test suite for non-Linux systems
- Add `spead2.send.trollius.UdpStream.async_flush()`

Version 0.4.1

- Add C++ version of spead2_recv, a more fully-featured alternative to test_recv
- **backwards-incompatible change**: Add *ring_heaps* parameter to `ring_stream` constructor. Code that specifies the *contiguous_only* parameter will need to be modified since the position has changed. Python code is unaffected.
- Increased the default for *ring_heaps* from 2 (previously hardcoded) to 4 to improve throughput for small heaps.
- Add support for user to provide the socket for UDP communications. This allows socket options to be set by the user, for example, to configure multicast.
- Force numpy>=1.9.2 to avoid a numpy [bug](https://github.com/numpy/numpy/issues/5356).
- Add experimental support for receiving packets via netmap
- Improved receive performance on Linux, particularly for small packets, using [recvmmsg](http://linux.die.net/man/2/recvmmsg).

Version 0.4.0

- Enforce ASCII encoding on descriptor fields.
- Warn if a heap is dropped due to being incomplete.
- Add –ring option to C++ spead2_bench to test ringbuffer performance.
- Reading from a memory buffer (e.g. with `add_buffer_reader()`) is now reliable, instead of dropping heaps if the consumer doesn't keep up (heaps can still be dropped if packets extracted from the buffer are out-of-order, but it is deterministic).
- The receive ringbuffer now has a fixed size (2), and pushes are blocking. The result is lower memory usage, and it is no longer necessary to pass a large *max_heaps* value to deal with the consumer not always keeping up. Instead, it may be necessary to increase the socket buffer size.
- **backwards-incompatible change**: Calling `spead2::recv::ring_stream::stop()` now discards remaining partial heaps instead of adding them to the ringbuffer. This only affects the C++ API, because the Python API does not provide any access to partial heaps anyway.
- **backwards-incompatible change**: A heap with a stop flag is swallowed rather than passed to `heap_ready()` (see issue [#29](https://github.com/ska-sa/spead2/issues/29)).

Version 0.3.0

This release contains a number of backwards-incompatible changes in the Python bindings, although most uses will probably not notice:

- When a received character array is returned as a string, it is now of type `str` (previously it was `unicode` in Python 2).
- An array of characters with a numpy descriptor with type *S1* will no longer automatically be turned back into a string. Only using a format of *[('c', 8)]* will do so.
- The *c* format code may now only be used with a length of 8.
- When sending, values will now always be converted to a numpy array first, even if this isn't the final representation that will be put on the network. This may lead to some subtle changes in behaviour.
- The *BUG_COMPAT_NO_SCALAR_NUMPY* introduced in 0.2.2 has been removed. Now, specifying an old-style format will always use that format at the protocol level, rather than replacing it with a numpy descriptor.

There are also some other bug-fixes and improvements:

- Fix incorrect warnings about send buffer size.
- Added –descriptors option to spead2_recv.py.
- The *dtype* argument to `spead2.ItemGroup.add_item()` is now optional, removing the need to specify *dtype=None* when passing a format.

Version 0.2.2

- Workaround for a PySPEAD bug that would cause PySPEAD to fail if sent a simple scalar value. The user must still specify scalars with a format rather than a dtype to make things work.

Version 0.2.1

- Fix compilation on OS X again. The extension binary will be slightly larger as a result, but still much smaller than before 0.2.0.

Version 0.2.0

- **backwards-incompatible change**: for sending, the heap count is now tracked internally by the stream, rather than an attribute of the heap. This affects both C++ and Python bindings, although Python code that always uses *HeapGenerator* rather than directly creating heaps will not be affected.
- The *HeapGenerator* is extended to allow items to be added to an existing heap and to give finer control over whether descriptors and/or values are put in the heap.
- Fixes a bug that caused some values to be cast to non-native endian.
- Added overloaded equality tests on Flavour objects.
- Strip the extension binary to massively reduce its size

Version 0.1.2

- Coerce values to int for legacy 'u' and 'i' fields
- Fix flavour selection in example code

Version 0.1.1

- Fixes to support OS X

Version 0.1.0

- First public release

CHAPTER

ELEVEN

LICENSE

This program is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this program. If not, see http://www.gnu.org/licenses/.

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

spead2, 7
spead2.recv.stream_stat_indices, 19

INDEX

Symbols

-C
    command line option, 100
-D
    command line option, 100
-N
    command line option, 100
--count
    command line option, 100
--direct-io
    command line option, 100

A

add_buffer_reader() (*spead2.recv.Stream method*), 13
add_descriptor() (*spead2.send.Heap method*), 21
add_end() (*spead2.send.Heap method*), 21
add_free_chunk() (*spead2.recv.ChunkRingStream method*), 35
add_free_chunk() (*spead2.recv.ChunkStreamRingGroup method*), 36
add_inproc_reader() (*spead2.recv.Stream method*), 15
add_item() (*spead2.ItemGroup method*), 10
add_item() (*spead2.send.Heap method*), 21
add_packet() (*spead2.InprocQueue method*), 27
add_producer() (*spead2.recv.ChunkRingbuffer method*), 34
add_start() (*spead2.send.Heap method*), 21
add_stat() (*spead2.recv.StreamConfig method*), 12
add_tcp_reader() (*spead2.recv.Stream method*), 14
add_to_heap() (*spead2.send.HeapGenerator method*), 20
add_udp_ibv_reader() (*spead2.recv.Stream method*), 30
add_udp_pcap_file_reader() (*spead2.recv.Stream method*), 15
add_udp_reader() (*spead2.recv.Stream method*), 13
async_flush() (*spead2.send.asyncio.AsyncStream method*), 26
async_send_heap() (*spead2.send.asyncio.AsyncStream method*), 25
async_send_heaps() (*spead2.send.asyncio.AsyncStream method*), 26
AUTO (*spead2.send.RateMethod attribute*), 20

B

BATCHES (*in module spead2.recv.stream_stat_indices*), 19
built-in function
    spead2.recv.Heap.is_end_of_stream(), 11
    spead2.recv.Heap.is_start_of_stream(), 11
    spead2.recv.IncompleteHeap.is_end_of_stream(), 18
    spead2.recv.IncompleteHeap.is_start_of_stream(), 18

C

capacity() (*spead2.recv.Stream.Ringbuffer method*), 18
chunk_allocate_function (*C++ type*), 74
chunk_id (*spead2.recv.Chunk attribute*), 32
chunk_place_data (*in module spead2.recv.numba*), 32
ChunkRingbuffer (*class in spead2.recv.asyncio*), 34
cnt (*spead2.recv.Heap attribute*), 11
cnt (*spead2.recv.IncompleteHeap attribute*), 17
combine() (*spead2.recv.StreamStatConfig method*), 18
command line option
    -C, 100
    -D, 100
    -N, 100
    --count, 100
    --direct-io, 100
compatible_shape() (*spead2.Descriptor method*), 9
config (*spead2.recv.Stream attribute*), 13
config (*spead2.recv.StreamStats attribute*), 18
connect() (*spead2.send.asyncio.TcpStream class method*), 26
COUNTER (*spead2.recv.StreamStatConfig.Mode attribute*), 18

D

data (*spead2.recv.Chunk attribute*), 32
data_fd (*spead2.recv.ChunkRingbuffer attribute*), 33
data_ringbuffer (*spead2.recv.ChunkRingStream attribute*), 35
data_ringbuffer (*spead2.recv.ChunkStreamRingGroup attribute*), 36
Descriptor (*class in spead2*), 9
disable_packet_presence() (*spead2.recv.ChunkStreamConfig method*), 33
dynamic_shape() (*spead2.Descriptor method*), 9

E

emplace_back() (*spead2.recv.ChunkStreamRingGroup method*), 36
empty() (*spead2.recv.ChunkRingbuffer method*), 33
enable_packet_presence() (*spead2.recv.ChunkStreamConfig method*), 33
environment variable
    M_MMAP_THRESHOLD, 96
    OMP_NUM_THREADS, 115
    PKG_CONFIG_PATH, 37
    SKIP=pip-compile, 113
    SPEAD2_IBV_COMP_VECTOR, 30
    SPEAD2_IBV_INTERFACE, 30
extra (*spead2.recv.Chunk attribute*), 32

F

fd (*spead2.recv.Stream attribute*), 15
Flavour (*built-in class*), 7
flavour (*spead2.recv.Heap attribute*), 11
flavour (*spead2.recv.IncompleteHeap attribute*), 17
flush() (*spead2.send.asyncio.AsyncStream method*), 26
free_ringbuffer (*spead2.recv.ChunkRingStream attribute*), 35
free_ringbuffer (*spead2.recv.ChunkStreamRingGroup attribute*), 36
full() (*spead2.recv.ChunkRingbuffer method*), 33

G

get() (*spead2.recv.asyncio.ChunkRingbuffer method*), 34
get() (*spead2.recv.asyncio.Stream method*), 16
get() (*spead2.recv.ChunkRingbuffer method*), 34
get() (*spead2.recv.Stream method*), 15
get_end() (*spead2.send.HeapGenerator method*), 21
get_heap() (*spead2.send.HeapGenerator method*), 21
get_nowait() (*spead2.recv.ChunkRingbuffer method*), 34
get_nowait() (*spead2.recv.Stream method*), 15
get_start() (*spead2.send.HeapGenerator method*), 21
get_stat_index() (*spead2.recv.StreamConfig method*), 12
getvalue() (*spead2.send.BytesStream method*), 25

H

heap_length (*spead2.recv.IncompleteHeap attribute*), 17
HeapGenerator (*class in spead2.send*), 20
HEAPS (*in module spead2.recv.stream_stat_indices*), 19
HW (*spead2.send.RateMethod attribute*), 20

I

ids() (*spead2.ItemGroup method*), 10
INCOMPLETE_HEAPS_EVICTED (*in module spead2.recv.stream_stat_indices*), 19
INCOMPLETE_HEAPS_FLUSHED (*in module spead2.recv.stream_stat_indices*), 19
InprocStream (*class in spead2.send.asyncio*), 28
intp_to_voidptr() (*in module spead2.numba*), 32
is_variable_size() (*spead2.Descriptor method*), 9
Item (*class in spead2*), 9
ItemGroup (*class in spead2*), 10
items() (*spead2.ItemGroup method*), 10
itemsize_bits (*spead2.Descriptor attribute*), 9

K

keys() (*spead2.ItemGroup method*), 10

L

LOSSLESS (*spead2.recv.ChunkStreamGroupConfig.EvictionMode attribute*), 35
LOSSY (*spead2.recv.ChunkStreamGroupConfig.EvictionMode attribute*), 35

M

M_MMAP_THRESHOLD, 96
MAX_BATCH (*in module spead2.recv.stream_stat_indices*), 19
MAXIMUM (*spead2.recv.StreamStatConfig.Mode attribute*), 18
maxsize (*spead2.recv.ChunkRingbuffer attribute*), 33
mode (*spead2.recv.StreamStatConfig attribute*), 18
module
    spead2, 7
    spead2.recv.stream_stat_indices, 19

N

name (*spead2.recv.StreamStatConfig attribute*), 18
next_stat_index() (*spead2.recv.StreamConfig method*), 12
num_substreams (*spead2.send.SyncStream attribute*), 22

O

OMP_NUM_THREADS, 115

P

packet_presence_payload_size (*spead2.recv.ChunkStreamConfig attribute*), 33
PACKETS (*in module spead2.recv.stream_stat_indices*), 19
payload_ranges (*spead2.recv.IncompleteHeap attribute*), 18
PKG_CONFIG_PATH, 37
present (*spead2.recv.Chunk attribute*), 32
put() (*spead2.recv.asyncio.ChunkRingbuffer method*), 34
put() (*spead2.recv.ChunkRingbuffer method*), 34
put_nowait() (*spead2.recv.ChunkRingbuffer method*), 34
Python Enhancement Proposals
    PEP 585, 127
    PEP 604, 127
    PEP 613, 127
    PEP 685, 126
    PEP 703, 116

Q

qsize() (*spead2.recv.ChunkRingbuffer method*), 33
queues (*spead2.send.InprocStream attribute*), 28

R

received_length (*spead2.recv.IncompleteHeap attribute*), 17
remove_producer() (*spead2.recv.ChunkRingbuffer method*), 34
repeat_pointers (*spead2.send.Heap attribute*), 21
ring_config (*spead2.recv.Stream attribute*), 13
ringbuffer (*spead2.recv.Stream attribute*), 15
ROUND_ROBIN (*spead2.send.GroupMode attribute*), 26

S

SEARCH_DIST (*in module spead2.recv.stream_stat_indices*), 19
send_heap() (*spead2.send.SyncStream method*), 22
send_heaps() (*spead2.send.SyncStream method*), 22
SERIAL (*spead2.send.GroupMode attribute*), 26
set_affinity() (*spead2.ThreadPool static method*), 11
set_cnt_sequence() (*spead2.send.SyncStream method*), 22
SINGLE_PACKET_HEAPS (*in module spead2.recv.stream_stat_indices*), 19
size() (*spead2.recv.Stream.Ringbuffer method*), 18
SKIP=pip-compile, 113
space_fd (*spead2.recv.ChunkRingbuffer attribute*), 33
spead2
    module, 7
spead2.InprocQueue (*built-in class*), 27
spead2.MemoryPool (*built-in class*), 17
spead2.MmapAllocator (*built-in class*), 16
spead2.recv.asyncio.Stream (*built-in class*), 16
spead2.recv.Chunk (*built-in class*), 32
spead2.recv.ChunkRingbuffer (*built-in class*), 33
spead2.recv.ChunkRingStream (*built-in class*), 35
spead2.recv.ChunkStreamConfig (*built-in class*), 32
spead2.recv.ChunkStreamConfig.DEFAULT_MAX_CHUNKS (*built-in variable*), 33
spead2.recv.ChunkStreamGroupConfig (*built-in class*), 35
spead2.recv.ChunkStreamGroupConfig.EvictionMode (*built-in class*), 35
spead2.recv.ChunkStreamGroupMember (*built-in class*), 36
spead2.recv.ChunkStreamRingGroup (*built-in class*), 36
spead2.recv.Heap (*built-in class*), 11
spead2.recv.Heap.is_end_of_stream()
    built-in function, 11
spead2.recv.Heap.is_start_of_stream()
    built-in function, 11
spead2.recv.IncompleteHeap (*built-in class*), 17
spead2.recv.IncompleteHeap.is_end_of_stream()
    built-in function, 18
spead2.recv.IncompleteHeap.is_start_of_stream()
    built-in function, 18
spead2.recv.RingStreamConfig (*built-in class*), 13
spead2.recv.Stream (*built-in class*), 13
spead2.recv.Stream.Ringbuffer (*built-in class*), 18
spead2.recv.stream_stat_indices
    module, 19
spead2.recv.StreamConfig (*built-in class*), 12
spead2.recv.StreamStatConfig (*built-in class*), 18
spead2.recv.StreamStatConfig.Mode (*built-in class*), 18
spead2.recv.StreamStats (*built-in class*), 18
spead2.recv.UdpIbvConfig (*built-in class*), 29
spead2.send.asyncio.AsyncStream (*built-in class*), 25
spead2.send.BytesStream (*built-in class*), 25
spead2.send.GroupMode (*built-in class*), 26
spead2.send.Heap (*built-in class*), 21
spead2.send.HeapReference (*built-in class*), 26
spead2.send.HeapReferenceList (*built-in class*), 27

spead2.send.InprocStream (*built-in class*), 27
spead2.send.RateMethod (*built-in class*), 20
spead2.send.StreamConfig (*built-in class*), 19
spead2.send.SyncStream (*built-in class*), 22
spead2.send.TcpStream (*built-in class*), 24
spead2.send.UdpIbvConfig (*built-in class*), 30
spead2.send.UdpIbvStream (*built-in class*), 31
spead2.send.UdpStream (*built-in class*), 23
spead2.ThreadPool (*built-in class*), 11
spead2::descriptor (*C++ struct*), 42
spead2::descriptor::description (*C++ member*), 42
spead2::descriptor::format (*C++ member*), 42
spead2::descriptor::id (*C++ member*), 42
spead2::descriptor::name (*C++ member*), 42
spead2::descriptor::numpy_header (*C++ member*), 42
spead2::descriptor::shape (*C++ member*), 42
spead2::inproc_queue (*C++ class*), 67
spead2::inproc_queue::add_packet (*C++ function*), 67
spead2::inproc_queue::stop (*C++ function*), 67
spead2::io_service_ref (*C++ class*), 38
spead2::io_service_ref::get_shared_thread_pool (*C++ function*), 38
spead2::io_service_ref::io_service_ref (*C++ function*), 38
spead2::io_service_ref::operator* (*C++ function*), 38
spead2::io_service_ref::operator-> (*C++ function*), 38
spead2::memory_allocator (*C++ class*), 50
spead2::memory_allocator::allocate (*C++ function*), 51
spead2::memory_allocator::deleter (*C++ class*), 51
spead2::memory_allocator::deleter::get_allocator (*C++ function*), 51
spead2::memory_allocator::deleter::get_user (*C++ function*), 51
spead2::memory_allocator::free (*C++ function*), 51
spead2::recv::chunk (*C++ class*), 74
spead2::recv::chunk::chunk_id (*C++ member*), 74
spead2::recv::chunk::data (*C++ member*), 74
spead2::recv::chunk::extra (*C++ member*), 74
spead2::recv::chunk::present (*C++ member*), 74
spead2::recv::chunk::present_size (*C++ member*), 74
spead2::recv::chunk::stream_id (*C++ member*), 74
spead2::recv::chunk_place_data (*C++ struct*), 73
spead2::recv::chunk_place_data::batch_stats (*C++ member*), 73
spead2::recv::chunk_place_data::chunk_id (*C++ member*), 73
spead2::recv::chunk_place_data::extra (*C++ member*), 73
spead2::recv::chunk_place_data::extra_offset (*C++ member*), 73
spead2::recv::chunk_place_data::extra_size (*C++ member*), 73
spead2::recv::chunk_place_data::heap_index (*C++ member*), 73
spead2::recv::chunk_place_data::heap_offset (*C++ member*), 73
spead2::recv::chunk_place_data::items (*C++ member*), 73
spead2::recv::chunk_place_data::packet (*C++ member*), 73
spead2::recv::chunk_place_data::packet_size (*C++ member*), 73
spead2::recv::chunk_place_function (*C++ type*), 73
spead2::recv::chunk_ready_function (*C++ type*), 74
spead2::recv::chunk_ring_stream (*C++ class*), 77
spead2::recv::chunk_ring_stream::chunk_ring_stream (*C++ function*), 77
spead2::recv::chunk_stream (*C++ class*), 76
spead2::recv::chunk_stream::chunk_stream (*C++ function*), 76
spead2::recv::chunk_stream::get_chunk_config (*C++ function*), 77
spead2::recv::chunk_stream::get_heap_metadata (*C++ function*), 77
spead2::recv::chunk_stream_config (*C++ class*), 74
spead2::recv::chunk_stream_config::default_max_chun (*C++ member*), 76
spead2::recv::chunk_stream_config::disable_packet_p (*C++ function*), 75
spead2::recv::chunk_stream_config::enable_packet_pr (*C++ function*), 75
spead2::recv::chunk_stream_config::get_allocate (*C++ function*), 75
spead2::recv::chunk_stream_config::get_items (*C++ function*), 75
spead2::recv::chunk_stream_config::get_max_chunks (*C++ function*), 75
spead2::recv::chunk_stream_config::get_max_heap_ext

(*C++ function*), 76

spead2::recv::chunk_stream_config::get_packet_presence_payload_size (*C++ function*), 75

spead2::recv::chunk_stream_config::get_place (*C++ function*), 75

spead2::recv::chunk_stream_config::get_ready (*C++ function*), 75

spead2::recv::chunk_stream_config::set_allocate (*C++ function*), 75

spead2::recv::chunk_stream_config::set_items (*C++ function*), 75

spead2::recv::chunk_stream_config::set_max_chunks (*C++ function*), 75

spead2::recv::chunk_stream_config::set_max_heap_extra (*C++ function*), 75

spead2::recv::chunk_stream_config::set_place (*C++ function*), 75

spead2::recv::chunk_stream_config::set_ready (*C++ function*), 75

spead2::recv::chunk_stream_group (*C++ class*), 81

spead2::recv::chunk_stream_group::begin (*C++ function*), 81

spead2::recv::chunk_stream_group::cbegin (*C++ function*), 82

spead2::recv::chunk_stream_group::cend (*C++ function*), 82

spead2::recv::chunk_stream_group::emplace_back (*C++ function*), 82

spead2::recv::chunk_stream_group::empty (*C++ function*), 81

spead2::recv::chunk_stream_group::end (*C++ function*), 81

spead2::recv::chunk_stream_group::operator[] (*C++ function*), 81

spead2::recv::chunk_stream_group::size (*C++ function*), 81

spead2::recv::chunk_stream_group::stop (*C++ function*), 82

spead2::recv::chunk_stream_group_config (*C++ class*), 80

spead2::recv::chunk_stream_group_config::default_max_chunks (*C++ member*), 81

spead2::recv::chunk_stream_group_config::eviction_mode (*C++ enum*), 80

spead2::recv::chunk_stream_group_config::eviction_mode::LOSSLESS (*C++ enumerator*), 80

spead2::recv::chunk_stream_group_config::eviction_mode::LOSSY (*C++ enumerator*), 80

spead2::recv::chunk_stream_group_config::get_allocate (*C++ function*), 81

spead2::recv::chunk_stream_group_config::get_eviction_mode (*C++ function*), 80

spead2::recv::chunk_stream_group_config::get_max_chunks

(*C++ function*), 80

spead2::recv::chunk_stream_group_config::get_ready (*C++ function*), 81

spead2::recv::chunk_stream_group_config::set_alloca (*C++ function*), 80

spead2::recv::chunk_stream_group_config::set_evicti (*C++ function*), 80

spead2::recv::chunk_stream_group_config::set_max_ch (*C++ function*), 80

spead2::recv::chunk_stream_group_config::set_ready (*C++ function*), 81

spead2::recv::chunk_stream_group_member (*C++ class*), 82

spead2::recv::chunk_stream_group_member::adjust_con (*C++ function*), 83

spead2::recv::chunk_stream_group_member::allocate (*C++ function*), 83

spead2::recv::chunk_stream_group_member::emplace_re (*C++ function*), 82

spead2::recv::chunk_stream_group_member::flush (*C++ function*), 83

spead2::recv::chunk_stream_group_member::flush_chun (*C++ function*), 83

spead2::recv::chunk_stream_group_member::get_chunk_ (*C++ function*), 83

spead2::recv::chunk_stream_group_member::get_config (*C++ function*), 83

spead2::recv::chunk_stream_group_member::get_heap_r (*C++ function*), 83

spead2::recv::chunk_stream_group_member::get_stats (*C++ function*), 83

spead2::recv::chunk_stream_group_member::start (*C++ function*), 82

spead2::recv::chunk_stream_group_member::stop (*C++ function*), 82

spead2::recv::chunk_stream_group_member::stop_rece (*C++ function*), 82

spead2::recv::chunk_stream_ring_group (*C++ class*), 84

spead2::recv::chunk_stream_ring_group::begin (*C++ function*), 84

spead2::recv::chunk_stream_ring_group::cbegin (*C++ function*), 84

spead2::recv::chunk_stream_ring_group::cend (*C++ function*), 84

spead2::recv::chunk_stream_ring_group::emplace_back (*C++ function*), 85

spead2::recv::chunk_stream_ring_group::empty (*C++ function*), 84

spead2::recv::chunk_stream_ring_group::end (*C++ function*), 84

spead2::recv::chunk_stream_ring_group::operator[] (*C++ function*), 84

spead2::recv::chunk_stream_ring_group::size

(*C++ function*), 84
spead2::recv::chunk_stream_ring_group::stop (*C++ function*), 85
spead2::recv::heap (*C++ class*), 39
spead2::recv::heap::get_cnt (*C++ function*), 40
spead2::recv::heap::get_descriptors (*C++ function*), 40
spead2::recv::heap::get_flavour (*C++ function*), 40
spead2::recv::heap::get_items (*C++ function*), 40
spead2::recv::heap::get_payload (*C++ function*), 40
spead2::recv::heap::heap (*C++ function*), 40
spead2::recv::heap::is_ctrl_item (*C++ function*), 40
spead2::recv::heap::is_end_of_stream (*C++ function*), 40
spead2::recv::heap::is_start_of_stream (*C++ function*), 40
spead2::recv::heap::to_descriptor (*C++ function*), 40
spead2::recv::incomplete_heap (*C++ class*), 40
spead2::recv::incomplete_heap::get_cnt (*C++ function*), 41
spead2::recv::incomplete_heap::get_flavour (*C++ function*), 41
spead2::recv::incomplete_heap::get_heap_length (*C++ function*), 41
spead2::recv::incomplete_heap::get_items (*C++ function*), 41
spead2::recv::incomplete_heap::get_payload (*C++ function*), 41
spead2::recv::incomplete_heap::get_payload_ranges (*C++ function*), 41
spead2::recv::incomplete_heap::get_received_length (*C++ function*), 41
spead2::recv::incomplete_heap::incomplete_heap (*C++ function*), 41
spead2::recv::incomplete_heap::is_ctrl_item (*C++ function*), 41
spead2::recv::incomplete_heap::is_end_of_stream (*C++ function*), 41
spead2::recv::incomplete_heap::is_start_of_stream (*C++ function*), 41
spead2::recv::inproc_reader (*C++ class*), 68
spead2::recv::inproc_reader::inproc_reader (*C++ function*), 68
spead2::recv::item (*C++ struct*), 41
spead2::recv::item::id (*C++ member*), 42
spead2::recv::item::immediate_value (*C++ member*), 42
spead2::recv::item::is_immediate (*C++ member*), 42
spead2::recv::item::length (*C++ member*), 42
spead2::recv::item::ptr (*C++ member*), 42
spead2::recv::live_heap (*C++ class*), 39
spead2::recv::live_heap::get_bug_compat (*C++ function*), 39
spead2::recv::live_heap::get_cnt (*C++ function*), 39
spead2::recv::live_heap::is_complete (*C++ function*), 39
spead2::recv::live_heap::is_contiguous (*C++ function*), 39
spead2::recv::live_heap::is_end_of_stream (*C++ function*), 39
spead2::recv::mem_reader (*C++ class*), 49
spead2::recv::packet_memcpy_function (*C++ type*), 52
spead2::recv::ring_stream (*C++ class*), 46
spead2::recv::ring_stream::begin (*C++ function*), 47
spead2::recv::ring_stream::end (*C++ function*), 47
spead2::recv::ring_stream::get_ring_config (*C++ function*), 47
spead2::recv::ring_stream::pop (*C++ function*), 46
spead2::recv::ring_stream::pop_live (*C++ function*), 46
spead2::recv::ring_stream::ring_stream (*C++ function*), 46
spead2::recv::ring_stream::try_pop (*C++ function*), 46
spead2::recv::ring_stream::try_pop_live (*C++ function*), 47
spead2::recv::ring_stream_config (*C++ class*), 45
spead2::recv::ring_stream_config::get_contiguous_on (*C++ function*), 46
spead2::recv::ring_stream_config::get_heaps (*C++ function*), 46
spead2::recv::ring_stream_config::set_contiguous_on (*C++ function*), 46
spead2::recv::ring_stream_config::set_heaps (*C++ function*), 46
spead2::recv::stream (*C++ class*), 44
spead2::recv::stream::emplace_reader (*C++ function*), 45
spead2::recv::stream::flush (*C++ function*), 45
spead2::recv::stream::get_config (*C++ function*), 45
spead2::recv::stream::get_stats (*C++

function), 45
spead2::recv::stream::start (*C++ function*), 45
spead2::recv::stream::stop (*C++ function*), 45
spead2::recv::stream_config (*C++ class*), 43
spead2::recv::stream_config::add_stat (*C++ function*), 44
spead2::recv::stream_config::get_allow_out_of_order (*C++ function*), 44
spead2::recv::stream_config::get_allow_unsized_heaps (*C++ function*), 43
spead2::recv::stream_config::get_bug_compat (*C++ function*), 44
spead2::recv::stream_config::get_explicit_start (*C++ function*), 44
spead2::recv::stream_config::get_max_heaps (*C++ function*), 43
spead2::recv::stream_config::get_memcpy (*C++ function*), 43
spead2::recv::stream_config::get_memory_allocator (*C++ function*), 43
spead2::recv::stream_config::get_stat_index (*C++ function*), 44
spead2::recv::stream_config::get_stats (*C++ function*), 44
spead2::recv::stream_config::get_stop_on_stop_item (*C++ function*), 43
spead2::recv::stream_config::get_stream_id (*C++ function*), 44
spead2::recv::stream_config::get_substreams (*C++ function*), 43
spead2::recv::stream_config::next_stat_index (*C++ function*), 44
spead2::recv::stream_config::set_allow_out_of_order (*C++ function*), 44
spead2::recv::stream_config::set_allow_unsized_heaps (*C++ function*), 43
spead2::recv::stream_config::set_bug_compat (*C++ function*), 44
spead2::recv::stream_config::set_explicit_start (*C++ function*), 44
spead2::recv::stream_config::set_max_heaps (*C++ function*), 43
spead2::recv::stream_config::set_memcpy (*C++ function*), 43
spead2::recv::stream_config::set_memory_allocator (*C++ function*), 43
spead2::recv::stream_config::set_stop_on_stop_item (*C++ function*), 43
spead2::recv::stream_config::set_stream_id (*C++ function*), 44
spead2::recv::stream_config::set_substreams (*C++ function*), 43
spead2::recv::stream_stat_config (*C++ class*), 56
spead2::recv::stream_stat_config::combine (*C++ function*), 57
spead2::recv::stream_stat_config::get_mode (*C++ function*), 57
spead2::recv::stream_stat_config::get_name (*C++ function*), 57
spead2::recv::stream_stat_config::mode (*C++ enum*), 56
spead2::recv::stream_stat_config::mode::COUNTER (*C++ enumerator*), 56
spead2::recv::stream_stat_config::mode::MAXIMUM (*C++ enumerator*), 56
spead2::recv::stream_stat_indices (*C++ type*), 57
spead2::recv::stream_stat_indices::batches (*C++ member*), 57
spead2::recv::stream_stat_indices::custom (*C++ member*), 57
spead2::recv::stream_stat_indices::heaps (*C++ member*), 57
spead2::recv::stream_stat_indices::incomplete_heaps (*C++ member*), 57
spead2::recv::stream_stat_indices::incomplete_heaps (*C++ member*), 57
spead2::recv::stream_stat_indices::max_batch (*C++ member*), 57
spead2::recv::stream_stat_indices::packets (*C++ member*), 57
spead2::recv::stream_stat_indices::search_dist (*C++ member*), 57
spead2::recv::stream_stat_indices::single_packet_he (*C++ member*), 57
spead2::recv::stream_stat_indices::worker_blocked (*C++ member*), 57
spead2::recv::stream_stats (*C++ class*), 52
spead2::recv::stream_stats::at (*C++ function*), 54
spead2::recv::stream_stats::batches (*C++ member*), 56
spead2::recv::stream_stats::begin (*C++ function*), 54, 55
spead2::recv::stream_stats::cbegin (*C++ function*), 54
spead2::recv::stream_stats::cend (*C++ function*), 54
spead2::recv::stream_stats::const_iterator (*C++ type*), 53
spead2::recv::stream_stats::const_pointer (*C++ type*), 53
spead2::recv::stream_stats::const_reference (*C++ type*), 53
spead2::recv::stream_stats::const_reverse_iterator

(*C++ type*), 53
spead2::recv::stream_stats::count (*C++ function*), 55
spead2::recv::stream_stats::crbegin (*C++ function*), 55
spead2::recv::stream_stats::crend (*C++ function*), 55
spead2::recv::stream_stats::difference_type (*C++ type*), 53
spead2::recv::stream_stats::empty (*C++ function*), 54
spead2::recv::stream_stats::end (*C++ function*), 54, 55
spead2::recv::stream_stats::find (*C++ function*), 55
spead2::recv::stream_stats::get_config (*C++ function*), 53
spead2::recv::stream_stats::heaps (*C++ member*), 56
spead2::recv::stream_stats::incomplete_heaps_evicted (*C++ member*), 56
spead2::recv::stream_stats::incomplete_heaps_flushed (*C++ member*), 56
spead2::recv::stream_stats::iterator (*C++ type*), 53
spead2::recv::stream_stats::key_type (*C++ type*), 53
spead2::recv::stream_stats::mapped_type (*C++ type*), 53
spead2::recv::stream_stats::max_batch (*C++ member*), 56
spead2::recv::stream_stats::operator+ (*C++ function*), 55
spead2::recv::stream_stats::operator+= (*C++ function*), 55
spead2::recv::stream_stats::operator= (*C++ function*), 53
spead2::recv::stream_stats::operator[] (*C++ function*), 54
spead2::recv::stream_stats::packets (*C++ member*), 56
spead2::recv::stream_stats::pointer (*C++ type*), 53
spead2::recv::stream_stats::rbegin (*C++ function*), 55
spead2::recv::stream_stats::reference (*C++ type*), 53
spead2::recv::stream_stats::rend (*C++ function*), 55
spead2::recv::stream_stats::reverse_iterator (*C++ type*), 53
spead2::recv::stream_stats::search_dist (*C++ member*), 56
spead2::recv::stream_stats::single_packet_heaps (*C++ member*), 56
spead2::recv::stream_stats::size (*C++ function*), 54
spead2::recv::stream_stats::size_type (*C++ type*), 53
spead2::recv::stream_stats::stream_stats (*C++ function*), 53
spead2::recv::stream_stats::value_type (*C++ type*), 53
spead2::recv::stream_stats::worker_blocked (*C++ member*), 56
spead2::recv::tcp_reader (*C++ class*), 49
spead2::recv::tcp_reader::tcp_reader (*C++ function*), 49
spead2::recv::udp_ibv_config (*C++ class*), 68
spead2::recv::udp_ibv_config::add_endpoint (*C++ function*), 68
spead2::recv::udp_ibv_config::default_buffer_size (*C++ member*), 69
spead2::recv::udp_ibv_config::default_max_poll (*C++ member*), 69
spead2::recv::udp_ibv_config::default_max_size (*C++ member*), 69
spead2::recv::udp_ibv_config::get_buffer_size (*C++ function*), 69
spead2::recv::udp_ibv_config::get_comp_vector (*C++ function*), 69
spead2::recv::udp_ibv_config::get_endpoints (*C++ function*), 68
spead2::recv::udp_ibv_config::get_interface_address (*C++ function*), 69
spead2::recv::udp_ibv_config::get_max_poll (*C++ function*), 69
spead2::recv::udp_ibv_config::get_max_size (*C++ function*), 68
spead2::recv::udp_ibv_config::set_buffer_size (*C++ function*), 69
spead2::recv::udp_ibv_config::set_comp_vector (*C++ function*), 69
spead2::recv::udp_ibv_config::set_endpoints (*C++ function*), 68
spead2::recv::udp_ibv_config::set_interface_address (*C++ function*), 69
spead2::recv::udp_ibv_config::set_max_poll (*C++ function*), 69
spead2::recv::udp_ibv_config::set_max_size (*C++ function*), 68
spead2::recv::udp_ibv_reader (*C++ class*), 70
spead2::recv::udp_ibv_reader::udp_ibv_reader (*C++ function*), 70
spead2::recv::udp_pcap_file_reader (*C++ class*), 50

spead2::recv::udp_pcap_file_reader::udp_pcap_file_reader (C++ function), 50
spead2::recv::udp_reader (C++ class), 47
spead2::recv::udp_reader::udp_reader (C++ function), 47, 48
spead2::ringbuffer (C++ class), 77
spead2::ringbuffer::add_producer (C++ function), 79
spead2::ringbuffer::begin (C++ function), 79
spead2::ringbuffer::capacity (C++ function), 79
spead2::ringbuffer::emplace (C++ function), 78
spead2::ringbuffer::end (C++ function), 79
spead2::ringbuffer::get_data_sem (C++ function), 79
spead2::ringbuffer::get_space_sem (C++ function), 79
spead2::ringbuffer::pop (C++ function), 78
spead2::ringbuffer::push (C++ function), 78
spead2::ringbuffer::remove_producer (C++ function), 79
spead2::ringbuffer::size (C++ function), 79
spead2::ringbuffer::stop (C++ function), 79
spead2::ringbuffer::try_emplace (C++ function), 78
spead2::ringbuffer::try_pop (C++ function), 78
spead2::ringbuffer::try_push (C++ function), 78
spead2::ringbuffer_empty (C++ class), 79
spead2::ringbuffer_full (C++ class), 79
spead2::ringbuffer_stopped (C++ class), 80
spead2::send::group_mode (C++ enum), 61
spead2::send::group_mode::ROUND_ROBIN (C++ enumerator), 61
spead2::send::group_mode::SERIAL (C++ enumerator), 61
spead2::send::heap (C++ class), 57
spead2::send::heap::add_descriptor (C++ function), 58
spead2::send::heap::add_end (C++ function), 58
spead2::send::heap::add_item (C++ function), 58
spead2::send::heap::add_pointer (C++ function), 58
spead2::send::heap::add_start (C++ function), 58
spead2::send::heap::get_flavour (C++ function), 58
spead2::send::heap::get_item (C++ function), 58
spead2::send::heap::get_repeat_pointers (C++ function), 59
spead2::send::heap::heap (C++ function), 58
spead2::send::heap::item_handle (C++ type), 58
spead2::send::heap::set_repeat_pointers (C++ function), 58
spead2::send::heap_reference (C++ struct), 59
spead2::send::heap_reference::cnt (C++ member), 59
spead2::send::heap_reference::heap (C++ member), 59
spead2::send::heap_reference::heap_reference (C++ function), 59
spead2::send::heap_reference::rate (C++ member), 59
spead2::send::heap_reference::substream_index (C++ member), 59
spead2::send::inproc_stream (C++ class), 67
spead2::send::inproc_stream::get_queue (C++ function), 67
spead2::send::inproc_stream::inproc_stream (C++ function), 67
spead2::send::item (C++ struct), 59
spead2::send::item::allow_immediate (C++ member), 60
spead2::send::item::id (C++ member), 60
spead2::send::item::immediate (C++ member), 60
spead2::send::item::is_inline (C++ member), 60
spead2::send::item::item (C++ function), 59
spead2::send::item::length (C++ member), 60
spead2::send::item::ptr (C++ member), 60
spead2::send::stream (C++ class), 61
spead2::send::stream::async_send_heap (C++ function), 62
spead2::send::stream::async_send_heaps (C++ function), 62, 63
spead2::send::stream::completion_handler (C++ type), 61
spead2::send::stream::flush (C++ function), 63
spead2::send::stream::get_io_service (C++ function), 62
spead2::send::stream::get_num_substreams (C++ function), 63
spead2::send::stream::set_cnt_sequence (C++ function), 62
spead2::send::stream_config (C++ class), 60
spead2::send::stream_config::get_burst_rate (C++ function), 61
spead2::send::stream_config::get_burst_rate_ratio

(C++ function), 61
spead2::send::stream_config::get_burst_size (C++ function), 61
spead2::send::stream_config::get_max_heaps (C++ function), 61
spead2::send::stream_config::get_max_packet_size (C++ function), 60
spead2::send::stream_config::get_rate (C++ function), 60
spead2::send::stream_config::get_rate_method (C++ function), 61
spead2::send::stream_config::set_burst_rate_ratio (C++ function), 61
spead2::send::stream_config::set_burst_size (C++ function), 61
spead2::send::stream_config::set_max_heaps (C++ function), 61
spead2::send::stream_config::set_max_packet_size (C++ function), 60
spead2::send::stream_config::set_rate (C++ function), 60
spead2::send::stream_config::set_rate_method (C++ function), 61
spead2::send::streambuf_stream (C++ class), 66
spead2::send::streambuf_stream::streambuf_stream (C++ function), 67
spead2::send::tcp_stream (C++ class), 66
spead2::send::tcp_stream::tcp_stream (C++ function), 66
spead2::send::udp_ibv_config (C++ class), 70
spead2::send::udp_ibv_config::add_endpoint (C++ function), 71
spead2::send::udp_ibv_config::add_memory_region (C++ function), 70
spead2::send::udp_ibv_config::default_buffer_size (C++ member), 72
spead2::send::udp_ibv_config::default_max_poll (C++ member), 72
spead2::send::udp_ibv_config::get_buffer_size (C++ function), 71
spead2::send::udp_ibv_config::get_comp_vector (C++ function), 71
spead2::send::udp_ibv_config::get_endpoints (C++ function), 70
spead2::send::udp_ibv_config::get_interface_address (C++ function), 71
spead2::send::udp_ibv_config::get_max_poll (C++ function), 71
spead2::send::udp_ibv_config::get_memory_regions (C++ function), 70
spead2::send::udp_ibv_config::get_ttl (C++ function), 70
spead2::send::udp_ibv_config::set_buffer_size (C++ function), 71
spead2::send::udp_ibv_config::set_comp_vector (C++ function), 71
spead2::send::udp_ibv_config::set_endpoints (C++ function), 70
spead2::send::udp_ibv_config::set_interface_address (C++ function), 71
spead2::send::udp_ibv_config::set_max_poll (C++ function), 71
spead2::send::udp_ibv_config::set_memory_regions (C++ function), 70
spead2::send::udp_ibv_config::set_ttl (C++ function), 70
spead2::send::udp_ibv_stream (C++ class), 72
spead2::send::udp_ibv_stream::udp_ibv_stream (C++ function), 72
spead2::send::udp_stream (C++ class), 64
spead2::send::udp_stream::udp_stream (C++ function), 64--66
spead2::set_log_function (C++ function), 68
spead2::thread_pool (C++ class), 37
spead2::thread_pool::get_io_service (C++ function), 38
spead2::thread_pool::set_affinity (C++ function), 38
spead2::thread_pool::stop (C++ function), 38
spead2::thread_pool::thread_pool (C++ function), 38
SPEAD2_IBV_COMP_VECTOR, 30
SPEAD2_IBV_INTERFACE, 30
SPEAD2_MAJOR (C macro), 103
SPEAD2_MINOR (C macro), 103
SPEAD2_PATCH (C macro), 103
SPEAD2_VERSION (C macro), 103
start() (spead2.recv.Stream method), 15
stats (spead2.recv.Stream attribute), 15
stats (spead2.recv.StreamConfig attribute), 13
stop() (spead2.InprocQueue method), 27
stop() (spead2.recv.ChunkRingbuffer method), 34
stop() (spead2.recv.Stream method), 15
stop() (spead2.ThreadPool method), 11
stream_id (spead2.recv.Chunk attribute), 32
SW (spead2.send.RateMethod attribute), 20

U

update() (spead2.ItemGroup method), 10

V

value (spead2.Item attribute), 9
values() (spead2.ItemGroup method), 10
version (spead2.Item attribute), 10

W

warn_on_empty (*spead2.MemoryPool attribute*), 17
WORKER_BLOCKED (*in module spead2.recv.stream_stat_indices*), 19