spead2 Documentation

Release 1.2.2

SKA South Africa

May 19, 2017

Contents

Contents:

CHAPTER 1

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 trollius.

Preparation

spead2 requires a modern C++ compiler supporting C++11 (currently only GCC 4.8 and Clang 3.4 have been tried) as well as Boost (including compiled libraries). The Python bindings have additional dependencies — see below. At the moment only GNU/Linux has been tested but other POSIX-like systems should work too (OS X is tested occasionally).

There is optional support for netmap and ibverbs for higher performance. If the libraries (including development headers) libraries are installed, they will automatically be detected and used.

If you are installing spead2 from a git checkout, it is first necessary to run `./bootstrap.sh` to prepare the configure script and related files. When building from a packaged download this is not required.

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`.

Installing spead2 for Python

The only Python dependencies are numpy and six, although support for asynchronous I/O also requires trollius. Running the test suite additionally requires nose, decorator and netifaces, and some tests depend on PySPEAD (they will be skipped if it is not installed). It is also necessary to have the development headers for Python, and Boost.Python.

To install (which will automatically pull in the mandatory dependencies), run:

```
./setup.py install
```

Other standard methods for installing Python packages should work too.

Installing spead2 for C++

The C++ API uses the standard autoconf installation flow i.e.:

```
./configure [options]
make
make install
```

For generic help with configuration, see `INSTALL` in the top level of the source distribution. Optional features are autodetected by default, but can be disabled by passing options to **`configure`** (run `./configure -h` to see a list of options).

One option that may squeeze out a very small amount of extra performance is `--enable-lto` to enable link-time optimization. Up to version 1.2.0 this was enabled by default, but it has been disabled because it often needs other compiler or OS-specific configuration to make it work. For GCC, typical usage is

```
./configure --enable-lto AR=gcc-ar RANLIB=gcc-ranlib
```

The installation will install some benchmark tools, a static library, and the header files. At the moment there is no intention to create a shared library, because the ABI is not stable.

CHAPTER 2

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.

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.

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, and enforcement may differ between Python 2 and 3. 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, one case is 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.

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`.

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()*.

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 in the way the parameters combine, 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'

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 the new value will not be transmitted. In this case, either manually increment the version number, or reassign the value.

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*)

Update the item descriptors and items from an incoming heap.

Parameters **heap** (*spead2.recv.Heap*) – Incoming heap

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

Return type dict

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).

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

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*.

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.

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.

Blocking receive

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

class spead2.recv.**Stream**(*thread_pool*, *bug_compat=0*, *max_heaps=4*, *ring_heaps=4*)

Parameters

- **thread_pool** (spead2.ThreadPool) – Thread pool handling the I/O
- **bug_compat** (*int*) – Bug compatibility flags (see *SPEAD flavours*)
- **max_heaps** (*int*) – The number of partial heaps that can be live at one time. This affects how intermingled heaps can be (due to out-of-order packet delivery) before heaps get dropped.
- **ring_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.

set_memory_allocator(*allocator*)
Set or change the memory allocator for a stream. See *Memory allocators* for details.

Parameters **pool** (spead2.MemoryAllocator) – New memory allocator

set_memcpy(*id*)
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.

Parameters **id** ({MEMCPY_STD, MEMCPY_NONTEMPORAL}) – Identifier for the copy function

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.
- **socket** (*socket.socket*) – If specified, this socket is used rather than a new one. The socket must be open but unbound. The caller must not use this socket any further, although it is not necessary to keep it alive. This is mainly useful for fine-tuning socket options such as multicast subscriptions.

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 with multicast (IPv4 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_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.

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`.

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.

Asynchronous receive

Asynchronous I/O is supported through trollius, which is a Python 2 backport of the Python 3 `asyncio` module. It can be combined with other asynchronous I/O frameworks like twisted.

class `spead2.recv.trollius.`**Stream**(**args*, ***kwargs*, *loop=None*)
See *spead2.recv.Stream* (the base class) for other constructor arguments.

Parameters **loop** – Default Trollius event loop for async operations. If not specified, uses the default Trollius event loop. Do not call *get_nowait* from the base class.

get(*loop=None*)
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.

Parameters **loop** – Trollius event loop to use, overriding constructor.

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*)
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.

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

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*)

Parameters

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

The constructor arguments are also instance attributes.

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=<Mock name='mock.Flavour()' id='140464881787960'>*)

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 resent 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.

Blocking send

class spead2.send.**UdpStream**(*thread_pool*, *hostname*, *port*, *config*, *buffer_size=DEFAULT_BUFFER_SIZE*, *socket=None*)

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

Parameters

- **thread_pool** (spead2.ThreadPool) – Thread pool handling the I/O
- **hostname** (*str*) – Peer hostname
- **port** (*int*) – Peer port
- **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.
- **socket** (*socket.socket*) – If specified, this socket is used rather than a new one. The socket must be open but unbound. The caller must not use this socket any further, although it is not necessary to keep it alive. This is mainly useful for fine-tuning socket options.

send_heap(*heap*, *cnt=-1*)

Sends a spead2.send.Heap to the peer, and wait for completion. There is currently no indication of whether it successfully arrived.

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.

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, *cnt* = 1.

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

class spead2.send.**UdpStream**(*thread_pool*, *multicast_group*, *port*, *config*, *buffer_size=DEFAULT_BUFFER_SIZE*, *ttl*)

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

Parameters

- **thread_pool** (spead2.ThreadPool) – Thread pool handling the I/O
- **multicast_group** (*str*) – Multicast group hostname/IP address
- **port** (*int*) – Destination port
- **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*, *multicast_group*, *port*, *config*, *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
- **multicast_group** (*str*) – Multicast group hostname/IP address
- **port** (*int*) – Destination port
- **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*, *multicast_group*, *port*, *config*, *buffer_size=524288*, *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
- **multicast_group** (*str*) – Multicast group hostname/IP address
- **port** (*int*) – Destination port
- **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.**BytesStream**(*thread_pool*, *config*)
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

send_heap(*heap*)
Appends a spead2.send.Heap to the memory buffer.

getvalue()
Return a copy of the memory buffer.

Return type bytes

Asynchronous send

As for asynchronous receives, asynchronous sends are managed by trollius. 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).

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[1]. To enable it, add -DSPEAD2_MAX_LOG_LEVEL=spead2::log_level::debug to the compiler options in setup.py.

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 Mellanox ConnectX®-3 NICs. It depends on device managed flow steering (DMFS), which may require using the Mellanox OFED version of libibverbs.

[1] Logging is done from separate C threads, which have to wait for Python's Global Interpreter Lock (GIL) in order to do logging.

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
- Only multicast is supported

Within these limitations, it is quite easy to take advantage of this faster code path. The main difficulty is that one *must* specify the IP address of the interface that will send or receive the packets. The netifaces module can help find the IP address for an interface by name.

System configuration

It is likely that some system configuration will be needed to allow this mode to work correctly. For 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
```

For more information, see the libvma documentation.

Receiving

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

spead2.recv.Stream.**add_udp_ibv_reader**(*endpoints*, *interface_address*, *max_size=DEFAULT_UDP_IBV_MAX_SIZE*, *buffer_size=DEFAULT_UDP_IBV_BUFFER_SIZE*, *comp_vector=0*, *max_poll=DEFAULT_UDP_IBV_MAX_POLL*)

Feed data from multicast IPv4 traffic. For backwards compatibility, one can also pass a single address and port as two separate arguments in place of *endpoints*.

Parameters

- **endpoints** (*list*) – List of 2-tuples, each containing a hostname/IP address the multicast group and the UDP port number.
- **interface_address** (*str*) – Hostname/IP address of the interface which will be subscribed
- **max_size** (*int*) – Maximum packet size that will be accepted
- **buffer_size** (*int*) – Requested memory allocation for work requests. Note that this is used to determine the number of packets to buffer; if the packets are smaller than *max_size*, then fewer bytes will be buffered.
- **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 readers 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).

Environment variables

An existing application can be forced to use ibverbs for all multicast 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.

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.trollius.UdpIbvStream` class, analogous to `spead2.send.trollius.UdpStream`.

class `spead2.send.`**`UdpIbvStream`**(*thread_pool*, *multicast_group*, *port*, *config*, *interface_address*, *buffer_size*, *ttl=1*, *comp_vector=0*, *max_poll=DEFAULT_MAX_POLL*)

Create a multicast IPv4 UDP stream using the ibverbs API

Parameters

- **`thread_pool`** (`spead2.ThreadPool`) – Thread pool handling the I/O
- **`multicast_group`** (*str*) – Multicast group hostname/IP address
- **`port`** (*int*) – Destination port
- **`config`** (*spead2.send.StreamConfig*) – Stream configuration
- **`interface_address`** (*str*) – Hostname/IP address of the interface which will be subscribed
- **`buffer_size`** (*int*) – Socket buffer size. A warning is logged if this size cannot be set due to OS limits.
- **`ttl`** (*int*) – Multicast TTL
- **`buffer_size`** – Requested memory allocation for work requests.
- **`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).

CHAPTER 3

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.

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.

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 `spead2::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.

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

Retrieve the embedded io_service.

void **stop**()

Shut down the thread pool.

Public Static Functions

void **set_affinity**(int *core*)

Set CPU affinity of current thread.

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.

Receiving

Heaps

Unlike the Python bindings, the C++ bindings expose two 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. Frozen heaps can be move-constructed from live heaps, which will typically be done in the callback.

class `spead2::recv::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. A complete heap is also contiguous, but not necessarily the other way around. Only contiguous heaps can be frozen.

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.

s_item_pointer_t **get_cnt**() **const**
: Retrieve the heap ID.

bug_compat_mask **get_bug_compat**() **const**
: Get protocol bug compatibility flags.

class spead2::recv::**heap**
Received heap that has been finalised.

Subclassed by spead2::recv::heap_wrapper

Public Functions

heap(*live_heap* &&*h*)
: Freeze a heap, which must satisfy *live_heap::is_contiguous*.

  The original heap is destroyed.

s_item_pointer_t **get_cnt**() **const**
: Get heap ID.

const flavour &**get_flavour**() **const**
: Get protocol flavour used.

const std::vector<*item*> &**get_items**() **const**
: Get the items from the heap.

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

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.

bool **is_start_of_stream**() **const**
Convenience function to check whether any of the items is a CTRL_STREAM_START.

struct spead2::recv::**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 spead2::**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.

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()`.

class `spead2::recv::`**`stream`**

Stream that is fed by subclasses of reader.

Unless otherwise specified, methods in stream_base may only be called while holding the strand contained in this class. The public interface functions must be called from outside the strand (and outside the threads associated with the io_service), but are not thread-safe relative to each other.

This class is thread-safe. This is achieved mostly by having operations run as completion handlers on a strand. The exception is *stop*, which uses a `once` to ensure that only the first call actually runs.

Inherits from spead2::recv::stream_base

Subclassed by callback_stream, recv_stream, spead2::recv::ring_stream_base

Public Functions

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

Add a new reader by passing its constructor arguments, excluding the initial *stream* argument.

void **`stop`**()

Stop the stream and block until all the readers have wound up.

After calling this there should be no more outstanding completion handlers in the thread pool.

In most cases subclasses should override *stop_received* rather than this function.

Protected Functions

void **`stop_received`**()

Shut down the stream.

This calls *flush*. Subclasses may override this to achieve additional effects, but must chain to the base implementation.

It is undefined what happens if add_packet is called after a stream is stopped.

void **`flush`**()

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

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>`.

template <typename *Ringbuffer* = ringbuffer<*live_heap*>>
class `spead2::recv::`**`ring_stream`**

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.

Inherits from spead2::recv::ring_stream_base

Readers

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

class spead2::recv::**udp_reader**

Asynchronous stream reader that receives packets over UDP.

Inherits from spead2::recv::udp_reader_base

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 multicast interface address (IPv4 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, use boost::asio::ip::address_v4::any(), or use the default constructor.

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_address: Address of the interface which should join the group

Exceptions

- std::invalid_argument: If *endpoint* is not an IPv4 multicast 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 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*, **const** boost::asio::ip::udp::endpoint &*endpoint*, std::size_t *max_size* = default_max_size, std::size_t *buffer_size* = default_buffer_size)

Constructor using an existing socket.

This allows socket options (e.g., multicast subscriptions) to be fine-tuned by the caller. The socket should not be bound. Note that there is no special handling for multicast addresses here.

Parameters

- `owner`: Owning stream
- `socket`: Existing socket which will be taken over. It must use the same I/O service as *owner*.
- `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.

class `spead2::recv::mem_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*.

Inherits from spead2::recv::reader

Subclassed by spead2::recv::buffer_reader

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 this can be any packet from the heap, so you must not rely on it being the initial packet.

class `spead2::`**`memory_allocator`**

Polymorphic class for managing memory allocations in a memory pool.

This can be overloaded to provide custom memory allocations.

Inherits from std::enable_shared_from_this< memory_allocator >

Subclassed by spead2::memory_pool, spead2::mmap_allocator, spead2::unittest::mock_allocator

Public Functions

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

Allocate *size* bytes of memory.

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

Return Pointer to newly allocated memory

Parameters

- `size`: Number of bytes to allocate
- `hint`: Usage-dependent extra information

Exceptions

- `std::bad_alloc`: if allocation failed

Private Functions

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

Free memory previously returned from *allocate*.

Parameters

- `ptr`: Value returned by *allocate*
- `user`: User-defined handle returned by *allocate*

Sending

Heaps

class `spead2::send::`**`heap`**

Heap that is constructed for transmission.

Subclassed by spead2::send::heap_wrapper

Public Functions

heap(**const** flavour &*flavour_* = flavour())
Constructor.

Parameters

- `flavour_`: SPEAD flavour that will be used to encode the heap

const flavour &**get_flavour**() **const**
Return flavour.

template <typename... *Args*>
void **add_item**(s_item_pointer_t *id*, Args&&... *args*)
Construct a new item.

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.

void **add_start**()
Add a start-of-stream control item.

void **add_end**()
Add an end-of-stream control item.

struct `spead2::send::item`
An item to be inserted into a heap.

An item does *not* own its memory.

Public Functions

item()
Default constructor.

This item has undefined values and is not usable.

item(s_item_pointer_t *id*, **const** void **ptr*, std::size_t *length*, bool *allow_immediate*)
Create an item referencing existing memory.

item(s_item_pointer_t *id*, s_item_pointer_t *immediate*)
Create an item with a value to be encoded as an immediate.

item(s_item_pointer_t *id*, **const** std::string &*value*, bool *allow_immediate*)
Construct an item referencing the data in a string.

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.

Streams

All stream types are derived from *spead2::send::stream* using the curiously recurring template pattern and implementing an `async_send_packet` function.

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

class spead2::send::**stream**
: Abstract base class for streams.

  Subclassed by spead2::send::stream_impl< Derived >, spead2::send::stream_impl< streambuf_stream >, spead2::send::stream_impl< udp_ibv_stream >, spead2::send::stream_impl< udp_stream >

Public Functions

boost::asio::io_service &**get_io_service**() **const**
: Retrieve the io_service used for processing the stream.

virtual void **set_cnt_sequence**(item_pointer_t *next*, item_pointer_t *step*) = 0
: 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.

virtual bool **async_send_heap**(**const** *heap* &*h*, *completion_handler* *handler*, s_item_pointer_t *cnt* = -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 due to insufficient space. The handler is called as soon as possible (from a thread running the io_service), with error code `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.

Return Value

- `false`: If the heap was immediately discarded
- `true`: If the heap was enqueued

virtual void **flush**() = 0

Block until all enqueued heaps have been sent.

This function is thread-safe, but can be live-locked if more heaps are added while it is running.

class `spead2::send::udp_stream`

Inherits from spead2::send::stream_impl< udp_stream >

Public Functions

udp_stream(boost::asio::io_service &*io_service*, **const** boost::asio::ip::udp::endpoint &*endpoint*, **const** stream_config &*config* = stream_config (), std::size_t *buffer_size* = default_buffer_size)

Constructor.

udp_stream(boost::asio::ip::udp::socket &&*socket*, **const** boost::asio::ip::udp::endpoint &*endpoint*, **const** stream_config &*config* = stream_config (), std::size_t *buffer_size* = default_buffer_size)

Constructor using an existing socket.

The socket must be open but not bound.

udp_stream(boost::asio::io_service &*io_service*, **const** boost::asio::ip::udp::endpoint &*endpoint*, **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
- `endpoint`: Multicast group and port
- `config`: Stream configuration
- `buffer_size`: Socket buffer size (0 for OS default)

- `ttl`: Maximum number of hops

Exceptions

- `std::invalid_argument`: if *endpoint* is not a multicast address

udp_stream(boost::asio::io_service &*io_service*, **const** boost::asio::ip::udp::endpoint &*endpoint*, **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
- `endpoint`: Multicast group and port
- `config`: Stream configuration
- `buffer_size`: Socket buffer size (0 for OS default)
- `ttl`: Maximum number of hops
- `interface_address`: Address of the outgoing interface

Exceptions

- `std::invalid_argument`: if *endpoint* is not an IPv4 multicast address
- `std::invalid_argument`: if *interface_address* is not an IPv4 address

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

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

See if_nametoindex(3)

Parameters

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

Exceptions

- `std::invalid_argument`: if *endpoint* is not an IPv6 multicast address

class `spead2::send::`**streambuf_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.

Inherits from spead2::send::stream_impl< streambuf_stream >

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

Public Functions

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

Constructor.

Logging

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

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

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 spead2::recv::**udp_ibv_reader**

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

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

Inherits from spead2::recv::udp_reader_base

Public Functions

udp_ibv_reader(*stream* &*owner*, **const** boost::asio::ip::udp::endpoint &*endpoint*, **const** boost::asio::ip::address &*interface_address*, std::size_t *max_size* = default_max_size, std::size_t *buffer_size* = default_buffer_size, int *comp_vector* = 0, int *max_poll* = default_max_poll)

Constructor.

Parameters

- `owner`: Owning stream
- `endpoint`: Multicast group and port
- `max_size`: Maximum packet size that will be accepted
- `buffer_size`: Requested memory allocation for work requests. Note that this is used to determine the number of packets to buffer; if the packets are smaller than *max_size*, then fewer bytes will be buffered.
- `interface_address`: Address of the interface which should join the group and listen for data
- `comp_vector`: 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 readers to be assigned sequential completion vectors and have them load-balanced, without concern for the number available.

- `max_poll`: 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).

Exceptions

- `std::invalid_argument`: If *endpoint* is not an IPv4 multicast address
- `std::invalid_argument`: If *interface_address* is not an IPv4 address

udp_ibv_reader(*stream* &*owner*, **const** std::vector<boost::asio::ip::udp::endpoint> &*endpoints*, **const** boost::asio::ip::address &*interface_address*, std::size_t *max_size* = default_max_size, std::size_t *buffer_size* = default_buffer_size, int *comp_vector* = 0, int *max_poll* = default_max_poll)

Constructor with multiple endpoints.

Parameters

- `owner`: Owning stream
- `endpoints`: Multicast groups and ports
- `max_size`: Maximum packet size that will be accepted
- `buffer_size`: Requested memory allocation for work requests. Note that this is used to determine the number of packets to buffer; if the packets are smaller than *max_size*, then fewer bytes will be buffered.
- `interface_address`: Address of the interface which should join the group and listen for data
- `comp_vector`: 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 readers to be assigned sequential completion vectors and have them load-balanced, without concern for the number available.
- `max_poll`: 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).

Exceptions

- `std::invalid_argument`: If any element of *endpoints* is not an IPv4 multicast address
- `std::invalid_argument`: If *interface_address* is not an IPv4 address

class `spead2::send::`**udp_ibv_stream**

Stream using Infiniband versions for acceleration.

Only IPv4 multicast with an explicit source address are supported.

Inherits from spead2::send::stream_impl< udp_ibv_stream >

Public Functions

udp_ibv_stream(boost::asio::io_service &*io_service*, **const** boost::asio::ip::udp::endpoint &*endpoint*, **const** stream_config &*config*, **const** boost::asio::ip::address &*interface_address*, std::size_t *buffer_size* = default_buffer_size, int *ttl* = 1, int *comp_vector* = 0, int *max_poll* = default_max_poll)

Constructor.

Parameters

- `io_service`: I/O service for sending data
- `endpoint`: Multicast group and port
- `config`: Stream configuration
- `interface_address`: Address of the outgoing interface
- `buffer_size`: Socket buffer size (0 for OS default)
- `ttl`: Maximum number of hops
- `comp_vector`: 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 readers to be assigned sequential completion vectors and have them load-balanced, without concern for the number available.
- `max_poll`: 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).

Exceptions

- `std::invalid_argument`: if *endpoint* is not an IPv4 multicast address
- `std::invalid_argument`: if *interface_address* is not an IPv4 address

Support for netmap

Introduction

As an experimental feature, it is possible to use the netmap framework to receive packets at a higher rate than is possible with the regular sockets API. This is particularly useful for small packets.

This is not for the faint of heart: it requires root access, it can easily hang the whole machine, and it imposes limitations, including:

- Only the C++ API is supported. If you need every drop of performance, you shouldn't be using Python anyway.
- Only Linux is currently tested. It should be theoretically possible to support FreeBSD, but you're on your own (patches welcome).
- Only IPv4 is supported.
- Fragmented IP packets, and IP headers with optional fields are not supported.
- Checksums are not validated (although possibly the NIC will check them).
- Only one reader is supported per network interface.
- All packets that arrive with the correct UDP port will be processed, regardless of destination address. This could mean, for example, that unrelated multicast streams will be processed even though they aren't wanted.

Usage

Once netmap is installed and the header file `net/netmap_user.h` is placed in a system include directory, pass **`NETMAP=1`** to **`make`** to include netmap support in the library.

Then, instead of *`spead2::recv::udp_reader`*, use *`spead2::recv::netmap_udp_reader`*.

class `spead2::recv::`**`netmap_udp_reader`**
Inherits from spead2::recv::reader

Public Functions

`netmap_udp_reader`(*stream* &*owner*, **const** std::string &*device*, uint16_t *port*)
Constructor.

Parameters

- `owner`: Owning stream
- `device`: Name of the network interface e.g., `eth0`
- `port`: UDP port number to listen to

CHAPTER 4

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 Intel CPU. You will need to consult other documentation to find equivalent commands for other systems.

System tuning

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 not 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.

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 QPI 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 `intel_idle.max_state=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.
- 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).

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.

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.

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.

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.

[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
[4] The UDP and IP header together add 28 bytes, bringing the IP packet to the conventional MTU of 1500 bytes.

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.

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.

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.

There are two versions of the benchmarking tool: one implemented in Python (**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.

On the receiver, pick a port number (which must be free for both TCP and UDP) and run

```
spead2_bench.py slave <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.

There are also separate **spead2_send** and **spead2_recv** (and Python equivalents) 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.

Kernel bypass APIs

There are two low-level kernel bypass networking APIs supported: *ibverbs* and *netmap*. These provide a zero-copy path from the NIC into the spead2 library, without the kernel being involved. This can make a huge performance difference, particularly for small packet sizes.

Of these, ibverbs is the recommended one: it can be used without being a root user, it is supported by both the Python and C++ APIs, can be used for both sending and receiving, can be used by multiple processes or streams simultaneously, and in simple cases requires only an environment variable to be set. The netmap support is no longer developed or tested.

These APIs are not free: they will only work with some NICs, require special kernel drivers and setup, have limitations in what networking features they can support, and require the application to specify which network device to use. Refer to the links above for more details.

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.

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 (C++)

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*.

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.

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 5

Other tools

mcdump

mcdump is a tool similar to tcpdump, but specialised for high-speed capture of multicast UDP traffic using hardware that supports the Infiniband Verbs API. It has only been tested on Mellanox ConnectX-3 NICs. Like gulp, it uses a separate thread for disk I/O and CPU core affinity to achieve reliable performance.

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.

Installation

The tool is automatically compiled and installed with spead2, provided that libiverbs 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.

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.

Unfortunately, unlike tcpdump, it is not possible to tell 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.

Limitations

- Packets are not timestamped (they all have a zero timestamp in the file).
- Only IPv4 is supported.

CHAPTER 6

Changelog

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 7

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 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

Symbols

A

C

D

E

F

G

H

I

K

M

P

S

U

V