Client:	SKA SA and NRF
Project:	CASPER
Type:	Data Interface Protocol Definition

SPEAD: Streaming Protocol for Exchanging Astronomical Data

Document number:	SSA4700-0000-001
Revision:	1
Classification:	Open Source, GPL
Author:	J. Manley, M. Welz, A. Parsons, S. Ratcliffe
Date:	2012/10/16

Document Approval

Function	Name	Designation	Affiliation	Date	Signature
Submitted by	R. van Rooyen	DBE DSP Specialist	SKA SA		
Approved by	M. Welz	DBE Software	SKA SA		
Approved by	J. Manley	DBE DSP Specialist	SKA SA		
Approved by	S. Ratcliffe	SPT Subsystem Manager	SKA SA		
Approved by	A. Parsons	PI	PAPER		
Accepted by	F. Kapp	DBE Subsystem Manager	SKA SA		

Document History

Revision	Date of Issue	ECN Number	Comments
A	2009/04/17	N/A	Misc changes made after meeting between DBE, CSS and DSG teams. Packet header field numbering altered, inclusion of higher resolution fields, etc
B	2009/04/30	N/A	Added Timestamp scale factor field (47), updated explanation and example meta data packets accordingly
C	2009/07/01	N/A	Conversion from memo to interface control document
D	2009/07/10	N/A	Renamed to NRF-FF-ICD-W-402
E	2009/07/13	N/A	Renamed to NRF-FF-ICD-F-402
F	2009/07/20	N/A	Made changes requested by Alan, retrospectively changed the version numbers to letters
G	2009/07/27	N/A	Renamed to K8000-0027V1-002 ICD
H	2009/08/05	N/A	Changes from internal review
I	2009/09/21	N/A	Started incorporating changes from OAR of Thomas
J	2009/10/27	N/A	Renamed to SPADE for public release. Incremented to version 3 of the protocol. Added data descriptors.
K	2009/11/09	N/A	Renamed to SPEAD. Added unified option and payload descriptor.
L	2010/03/21	N/A	Misc changes after discussions between J. Manley and A. Parsons.
M	2010/06/07	N/A	Document revision and renumbering of items.
N	2010/08/28	N/A	Updating for implementation changes and error correction.
O	2010/10/07	N/A	First review before document release.
1	2012/10/16	N/A	First revision release.

Document Software

Task	Package	Version
Stylesheet	katdoc	1.1.2
Text processor	LaTeX	3.1415926-1.40.10
Editor	Vim	7.2.445

Company Details

Name	SKA South Africa Office
Physical/Postal Address	Third Floor The Park Park Road Pinelands 7405
Tel.	+27 21 506 7300
Fax	+27 21 506 7375
Website	http://www.ska.ac.za/

Contents

List of Figures

List of Tables

List of Abbreviations

FF	Fringe Finder
ICD	Interface Control Document
IP	Internet Protocol
KAT	Karoo Array Telescope
KATCP	KAT Communication Protocol
TCP	Transmission Control Protocol

1 Applicable and Reference Documents

1.1 Applicable Documents

The following documents are applicable to the extent stated herein. In the event of conflict between the contents of the applicable documents and this document, the applicable documents shall take precedence.

- 1 - Simon Ratcliffe, *(https://github.com/ska-sa/PySPEAD)*, 2010.

1.2 Related Documents

The following documents are referenced in this document. In the event of conflict between the contents of the referenced documents and this document, this document shall take precedence.

[1] Simon Cross, Marc Welz, Jason Manley. Kat control protocol specification. http://casper.berkeley.edu/wiki/images/1/11/NRF-KAT7-6.0-IFCE-002-Rev4.pdf.

[2] Jason Manley, Marc Welz, Simon Ratcliffe. Kat-7 data interface. K0000-2006V1-05.

2 Glossary

TCP/IP	Transmission Control Protocol (TCP) is a set of rules (protocol) used along with the Internet Protocol (IP) to send data in the form of message units between computers over the Internet. While IP takes care of delivery of the data, TCP takes care of keeping track of the individual units of data (called packets) that a message is divided into for efficient routing through the Internet.
UDP	User Datagram Protocol (UDP) uses the IP to transport a data unit (called a datagram) from one computer to another, but UDP makes no guarantee about data delivery, nor does it provide sequencing of the packets. This means that it is up to the application program that uses UDP to make sure that the entire message has arrived and is in the correct order.
Datagram	A datagram is a self-contained, independent entity of data carrying sufficient information to be routed from the source to the destination computer without reliance on earlier exchanges between the source and destination computer and the transporting network.
Item	A variable transmitted using the SPEAD protocol.
ItemGroup	A collection of *Items* to be transmitted.
Heap	An *ItemGroup* packaged for transmission as UDP data packets.
ItemPointer	Meta data in the packet header containing information on how to unpack the received datagram.

3 Scope

This document describes a data stream format suitable for use by radio astronomy instruments. The data stream is distinct from the KATCP control protocol described in Reference [1].

Many instruments output data over Ethernet. These streams are often high speed and unsuitable for TCP/IP links. UDP is often preferred in the cases where partial data-loss can be accommodated. This document aims to standardise the output format of such UDP streams.

An essential feature of the data stream is that it is self describing. This self description extends to the content and interpretation of the optional fields in the packet header, as well as the data payload itself. The same receive and transmit code may be used for multiple instruments as the layout of the packets are well defined.

Emphasis is placed on lightweight implementation, suitable for data packetisation by hardware devices or by software processes, while retaining flexibility for complex, dynamically changing data exchanges. Descoping, by way of fixed packet format, is required for simple receivers.

Metadata and variable descriptors can be injected into the data stream at runtime and recorded or processed along with the primary data stream. It is also possible to record the raw data stream directly to disk for later parsing. Note: Data flow is unidirectional in a given SPEAD data stream.

The format of the application-layer packetised stream is described in this document. Lower speed and ad-hoc data products may be transferred using the TCP-based KATCP control interface [1].

4 Protocol Layers

The data stream format describes the application and presentation layers of the OSI model. Data transmission can take place over any medium—such as serial data lines, or Ethernet networks—and is generally transmitted via a network interface, or stored on disk.

The preferred interface for network communications consists of the following layers:

Application and Presentation :
The data format as described in this document.

Session and Transport :
UDP, noting that the receiving party cannot request retransmission of lost, duplicated or corrupt packets.

Network :
IP (Internet protocol).

Link :
Ethernet (100 GbE, 40 GbE, 10 GbE, GbE, or 100 Mb/s).

Physical :
Any Ethernet medium, such as CFP, QSFP+, SFP+, CX4 or other 10 Gigabit Ethernet cable, Cat6, Cat5e, etc.

5 Definition

The SPEAD protocol is designed to propagate changes in a set of variables from one location to another. The origin and use of these variables are application dependent and is, as such, beyond the scope of this document. See KAT-7 data interface document for KAT-7 example implementation [2].

SPEAD uses the UDP protocol to transmit data packets called datagrams from one computer to another. Each user-created datagram consists of a SPEAD header, three or more optional *ItemPointer* fields and a data payload. While SPEAD can cope with lost or missing packets, it has no mechanism to request resending of missing data.

Different flavours of SPEAD can be created by specifying the number of addressable variables and total variable-space size. The SPEAD flavour is expressed as SPEAD-*XX*-*YY* where *XX* is the *ItemPointer* bit length and *YY* is the heap address space; *XX* is always greater than or equal to *YY*. This format is self-defining and these widths are configurable in the SPEAD header. Once specified, the allocated SPEAD sizes (*XX* and *YY*) should remain static for a given implementation.

SPEAD-*64-40* is currently deployed, allowing for data heaps of up to 1T/B (40 bit) and up to 2^{23} (8388608) variables (1-bit is reserved for addressing mode). The *Heap* size limit (*YY*) does not limit the total size of the SPEAD stream, it merely constrains the total number of variables *Item*s that can be changed simultaneously.

6 Data Overview

Each variable is associated with an *Item*. The *Item* contains:

- the variable's value, which can be a single-valued data product, multi-dimensional data, or can point to other *Item*s or multiples of other *Item*s, thereby allowing hierarchical data structures;
- associated metadata, i.e. the variable name, description, type and shape;
- a numerical identifier.

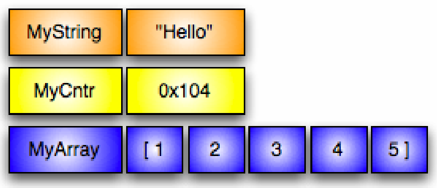

Figure 1: Consider a grouping of data that we might wish to transmit. It has multiple data elements, including a string "MyString", an integer counter "MyCntr", and an array of 32-bit unsigned integers called "MyArray".

*Item*s are collected into *ItemGroup*s. An *ItemGroup* can be modified by the addition of *Item*s, and by the modification of an *Item*'s value or its metadata. Once *Item*s in an *ItemGroup* have been modified to a selfconsistent state, the change in the state of the *ItemGroup* can be encoded as a *Heap*.

The *Heap* is subdivided into a collection of data packets and sent as a stream of UDP datagrams from the transmitter to the receiver. *ItemPointer*s in the packet header point to an address within an assembled *Heap* of data where the *Item*'s data-products are stored. This allows for very large data lengths for a given *Item*, which is unrestricted by the packet length. On the receiver, the *Heap* is reassembled and then unpacked as an *ItemGroup*. All *Item*s can be automatically unpacked and interpreted before presentation to the user's application.

*ItemPointer*s can be used as either an immediate address or as an absolute address. Immediate addressing refers to using the *ItemPointer* as a value instead of a pointer. This is more efficient for *Item*s which contain either character or single-valued items. Absolute addressing refers to using the *ItemPointer* as an address within an assembled *Heap*. This address is a byte offset within the *Heap* where the *Item*'s data-products are stored.

7 Data Flow

A *Heap* consists of a header section, followed by a payload section. The header section is a list of *ItemPointer*s, while the payload contains *Item*s' values.

The process of encoding changes in *Item*s as a *Heap* proceeds as follows:

1. For each *Item* whose value has changed, the binary representation of the new value is appended to the *Heap* payload, and an *ItemPointer* is appended to the *Heap* header. An *ItemPointer* encodes the *Item*'s numerical identifier, and a byte offset into the *Heap*'s payload where the *Item*'s value begins. For the special case where an *Item*'s value fits within the field allocated for the *ItemPointer* address, the value is not appended to the *Heap*, but is instead placed in the address field, and a one-bit IMMEDIATE flag in the *ItemPointer* is set to indicate that the value can be accessed directly as illustrated in Figure 2.

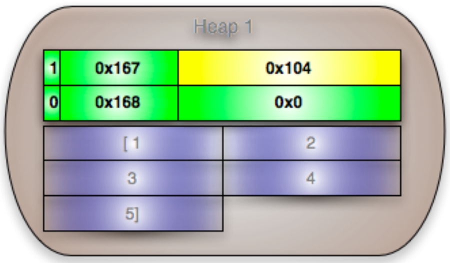

Figure 2: An example *Heap* of *Item*s to transmit: user-defined *Item* with identifier 0x167 has an integer value of 0x104 and is included directly into the header (with the IMMEDIATE flag set to 1), while user-defined *Item* 0x168's value can be found at *Heap* offset 0x0.

2. An *ItemDescriptor* is generated for each new *Item* that has been added to the *ItemGroup*, that is, the first time this variable is created. An *ItemDescriptor* is also created when a specific *Item*'s structure has changed. The value of an *ItemDescriptor* is itself a *SpeadStream* that may contain such *Item*s as NAME, DESCRIPTION, TYPE, SHAPE, and ID. The binary representation of this *SpeadStream* is appended to the *Heap* payload, and a special *ItemPointer* is added to the header of the *Heap* that has its numerical identifier set to DESCRIPTOR_ID, and points to the start of the *ItemDescriptor*'s value in the *Heap* payload. The reader is referred to the section on *ItemDescriptor*s for more detail.

3. This *Heap* is now converted into a series of SPEAD packets. Each packet consists of a SPEAD header, a number of *ItemPointer*s, and a payload. With the exception of some *ItemPointer*s added to aid in packet decoding and reassembly, all *ItemPointer*s are drawn from the *Heap* header. Each packet payload contains a portion of the *Heap* payload. Taken together, the generated series of packets will contain the entirety of the *Heap*'s header and payload, along with a small number of IMMEDIATE *ItemPointer*s that allow each packet to be associated with reconstructing a single *Heap* on the receive side.

4. After packets pass through a physical transport layer – such as a network interface, POSIX pipe, file – a receiver sorts incoming packets by associated *Heap* and begins the process of reconstructing the *Heap*. Packet headers are concatenated into a *Heap* header, and packet payloads are concatenated in the correct order to reconstruct the *Heap* payload. Missing packets are flagged.

5. Next, each *ItemPointer* in the *Heap* header is processed, associating a numerical identifier with a the binary value. For DIRECT-addressed *ItemPointer*s, this involves seeking within the *Heap* payload to the specified address offset and extracting data from that point to the start of the next *ItemPointer*. For IMMEDIATE *ItemPointer*s, the binary value is extracted directly from the address field.

6. Finally, the receiver's *ItemGroup* associates each numerical identifier and binary value pair with an *Item* having the same identifier. All *ItemPointer*s are processed, creating new *Item*s when appropriate. Each binary value is then decoded to an actual value by using the structure information provided by the associated *ItemDescriptor*.

8 Packet Data Representation

Data is always transmitted and stored most significant bit first, that is, Internet Protocol standard byte ordering. Because this same ordering is used for on-disk storage, data can be streamed directly between the network and storage.

At a minimum, a SPEAD packet must contain the following, as illustrated in Figure 3:

- A SPEAD header
- Three *ItemPointer*s, in no particular order, describing:
  - the packet payload length to locate packet boundaries for on-disk storage,
  - the offset of this packet's contents within the *Heap*,
  - and a counter, timestamp, or other unique number indicating the *Heap* to which it belongs.

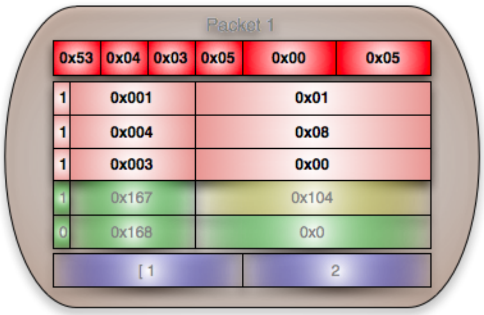

Figure 3: Example SPEAD-64-40 packet containing 5 *Item*s. This packet belongs to *Heap* 1 and has a payload that is 8 bytes long which starts at *Heap* offset 0x00 i.e. this is the first packet of the *Heap*.

8.1 SPEAD header

The SPEAD header is always 8 octets (64 bits) long. All packets have a header of the following form:

MSb					*LSb*
Magic # (8b)	Version (8b)	ItemPointerWidth (8b)	HeapAddrWidth (8b)	Reserved (16b)	# Items (16b)

MagicNumber : This is a constant value. All packets will start with the following 8 bit pattern: *(msb) 0101 0011 (lsb)* which has a numerical value of 0x53 and represents an ASCII *S*. This allows basic identification of the packet as a SPEAD packet.

Version : This field identifies the version of the packet format. Designed in case future packet structures differ. The version described here is 4.

ItemPointerWidth and HeapAddressWidth : These fields describe the number of bytes of each *Heap* address (*HeapAddressesWidth*) and the total size of an *ItemIdentifier*, the *ItemIdentifierWidth*. The sum of the two numbers indicate the size of an *ItemPointer*.

Initial implementations of SPEAD (SPEAD-64-40) have 64 bit *ItemPointer* fields, of which 40 bits address the data in a *Heap*, one bit chooses the addressing mode and the remaining 23 bits represent a numeric *ItemIdentifier*. HeapAddressWidth is thus set to 40bits/8bits = 5 bytes and ItemPointerWidth set to 24bits/8bits = 3 bytes, which includes the ItemPointerWidth which is the one bit, the MSB, always used for Addressing mode. This configuration defines the flavour of SPEAD and is referenced as SPEAD-64-40. These numbers need to remain static for a given configuration or implementation once the stream has started.

Reserved : This field should be ignored. The space is reserved for future allocation.

NumberOfItems : The value stored in this field indicates the number of *ItemPointer*s contained in this packet. *ItemPointer*s follow immediately after the SPEAD header. The packet payload follows directly after these *ItemPointer* fields. This field allows the offset of the payload to be determined.

8.2 *ItemPointers*: optional fields

*Item*s are used to transfer variables. Each packet can contain a variable number of data fields, called *ItemPointer*s, which follow directly after the SPEAD header. Each *ItemPointer* consists of some bits of *ItemIdentifier* and some bits of *ItemAddress*, a pointer to the location of the data in the *Heap*, in direct address mode, or the data itself, in immediate address mode.

*ItemPointer*s can have a user-defined length, which defines the flavour of the SPEAD protocol employed. See above description for more information on sizing these fields.

The number of *ItemPointer*s in a packet is defined by the *NumberOfItems* field in the SPEAD header, and should be 3 or more; a minimum of 3 are required by SPEAD.

ItemAddressMode : An *ItemPointer*'s lower bits, the *ItemAddress* field, normally store the *Heap* address where that *Item*'s data can be found. This is called absolute addressing mode.

Alternatively, the *ItemAddress* field can store the *Item*'s value. This is called immediate addressing mode.

Using immediate addressing allows for data widths of up to *HeapAddressWidth* bytes to be accommodated without incurring the additional overhead of a pointer. Whereas using a pointer in direct addressing mode allows for lengths of data longer than *HeapAddressWidth* bytes.

The addressing mode, and hence the interpretation of the *ItemAddress* field, is determined by the first bit in the *ItemPointer*. If this bit is clear, the *ItemAddress* field is interpreted as an index into the payload of the *Heap* and is a direct address. If this bit is set, the associated address is interpreted to immediately contain the value of the *Item* and is an immediate address.

ItemIdentifier : This field (23 bits for SPEAD-64-40) contains a numerical identifier that associates this *ItemPointer* with a corresponding *Item* whose value is being updated.

ItemAddress : The address (40 bits for SPEAD-64-40) can contain either a direct address into the *Heap* where the value for this *Item* is located, or it can contain the *Item*'s value. The interpretation of this field depends on how the *ItemAddressMode* field is set.

Example of an *ItemPointer* in SPEAD-64-40:

MSb		*LSb*
ItemAddressMode (1b)	ItemIdentifier (23b)	ItemAddress (40b)

If a packet is received that contains an unknown *Item*, it might be recorded and ignored or otherwise the receiver might request a metadata reissue to explain the content (see item 0x0005, the ITEM_DESCRIPTOR *Item*). In any case, the reception of unknown *Item*s should not be considered illegal and should be handled cleanly by a receiver.

Generally the *Item*s will be self defining and thus do not need strict identification in this document. However, it is necessary to define a few *Item*s which will be used for stream configuration and control.

The following table defines these standardised *ItemIdentifier*s and their meaning. Note that some of these require certain address modes.

Dec	*Hex*	*Description*	*Addr mode*
0	0x0000	NULL - Ignore	Ignored
1	0x0001	Heap counter	Immediate
2	0x0002	Heap size	Immediate
3	0x0003	Heap offset	Immediate
4	0x0004	Packet payload length	Immediate
5	0x0005	Item descriptor	Absolute
6	0x0006	Stream control	Immediate
16	0x0010	Item descriptor: name	Not specified
17	0x0011	Item descriptor: description	Not specified
18	0x0012	Item descriptor: shape	Not specified
19	0x0013	Item descriptor: type	Not specified
20	0x0014	Item descriptor: ID	Immediate
21	0x0015	Item descriptor: dtype	Absolute

The remaining *ItemIdentifier* address space is available for user specification. It is recommended that users' applications start at *Item* #1024.

Herewith follows an explanation for these standard *Item*s:

8.2.1 NULL

This *Item* is used for optional data padding and can be ignored by the receiver.

8.2.2 Heap counter

Required field.

Normally just a counter that increments with each *Heap* update. This number does not need to be continuous or even monotonic, but must be unique. It is used to identify the *Heap* to which a packet belongs.

Numerical Identifier : 0x0001

Name : heap_counter

Description : Identify to which *Heap* a packet belongs.

Type : unsigned integer, immediate addressing.

Shape : 1

8.2.3 Heap size

Optional *Item*.

It is highly recommended that this *Item* be issued for each *Heap* to allow receivers to detect when a *Heap* is complete (as opposed to waiting for a timeout or buffer overflow).

Numerical Identifier : 0x0002

Name : heap_size

Description : Specify the total size of the *Heap*, up-front, in bytes. Allows processing right after reception of last packet of the *Heap*.

Type : unsigned integer, immediate addressing.

Shape : 1

8.2.4 Heap offset

Required field.

The position relative to the start of this *Heap*, at which the payload of the current packet starts. It is used for re-assembling a *Heap* that was split across multiple packets. It is reset back to zero for the following *Heap*.

For example, in a correlator, this field would typically measure the byte offset from the start of an integration, as each integration would constitute another *Heap*.

Numerical Identifier : 0x0003

Name : heap_offset

Description : The position in bytes relative to the start of this *Heap* at which the payload of the current packet starts.

Type : unsigned integer, immediate addressing.

Shape : 1

8.2.5 Packet payload length

Required field.

This is the size in bytes of the packet payload length, excluding the header and item fields.

Numerical Identifier : 0x0004

Name : pkt_len

Description : The length in bytes of this packet's payload.

Type : unsigned integer, immediate addressing.

Shape : 1

8.2.6 Stream control

Optional *Item*.

> **Numerical Identifier** : 0x0006
>
> **Name** : stream_ctrl
>
> **Description** : Indicates the status of the SPEAD stream.
>
> **Type** : unsigned integer, immediate addressing.
>
> **Shape** : 1

The STREAM_CONTROL *Item* can indicate the one of the following status for a SPEAD stream:

0 : Stream start

1 : *ItemDescriptor* reissue (see section below on *ItemDescriptor*s)

2 : Stream stop

3 : *ItemDescriptor* update (*Item* has changed dimensions)

>=4 : Reserved for future use. Ignore.

8.3 *ItemDescriptors*

*ItemDescriptor*s are used to provide receiving clients with the metadata required to decode, interpret and unpack *Heap*s to form *ItemGroup*s as part of a particular SPEAD stream. In general, these packets will be sent at the start of a session before data packet transmissions begin.

ItemDescriptor updates can be issued at any time, including during a data stream. Such re-issuances are encouraged, so that if a receiver were to fail or lose state, it can recover and continue processing incoming data. Such packets may or may not contain a STREAM_CONTROL *Item* with the value of 1 (*ItemDescriptor* reissue).

Updates to *Item*s that are captured together with data, i.e. as part of the same *Heap*, are applied immediately to that captured *Heap* prior to unpacking or data interpretation. This allows the user to change object dimensions on-the-fly. Such data may or may not be accompanied by a STREAM_CONTROL *Item* with the value of 3 (*ItemDescriptor* update).

*ItemDescriptor*s are themselves complete SPEAD packets, including the SPEAD header (see Appendix for example usage). Each *ItemDescriptor* should contain at least the following *Item*s:

ItemDescriptorID : The numerical code used to identify the *Item* that we are describing. It has a bitwidth of $ItemPointerWidth - HeapAddressWidth - 1$.

ItemDescriptorName : A short string identifying the variable contained in this *Item*.

ItemDescriptorDescription : A human-readable string explaining the variable's use and packing scheme.

ItemDescriptorType : A binary-packed string representing the data type (signed or unsigned integer, float, string, boolean etc) and number of bits per value.

ItemDescriptorShape : A binary-packed string representing the size along each dimension.

ItemDescriptorDType : An optional string specifying the shape and type of this item using numpy dtype string syntax.

8.3.1 ItemDescriptorID

The value of this *ItemDescriptor Item* is the identifier of the *Item* that we are trying to describe in this *ItemDescriptor*.

Numerical Identifier : 0x0014

Name : descriptor_id

Description : The numerical code used to identify the *Item* described by this *ItemDescriptor*.

Type : unsigned integer, immediate addressing.

Shape : 1

8.3.2 ItemDescriptorName

Payload names are intended to be parsed and presented in the receiver namespace and thus should not include whitespace (including [LF], [FF] etc).

SPEAD does not mandate a maximum length for this field. It will be unpacked as the variable name by the receiver, so this should not be a verbose field. Use *ItemDescriptorDescription* for detailed explanations.

Numerical Identifier : 0x0010

Name : descriptor_name

Description : The variable name used to identify the *Item* described by this *ItemDescriptor*.

Type : string

Shape : *variable length, single dimension*

8.3.3 ItemDescriptorDescription

Numerical Identifier : 0x0011

Name : descriptor_description

Description : The human-readable description of the *Item* described by this *ItemDescriptor*.

Type : string

Shape : *variable length, single dimension*

8.3.4 ItemDescriptorType

Numerical Identifier : 0x0013

Name : descriptor_type

Description : Machine-parsable string detailing this *Item*'s binary packing.

Type : string

Shape : *variable length, single dimension*

This *Item* contains an unpack string telling the parser how to unpack the payload, and follows a C style printf convention, with the addition of referenced data types. The *ItemType*'s string payload can contain one or more unpack directives, with each directive referring to a consecutive data type in the payload. Each directive has the following form: *unpack_type[bit_length | ItemIdentifier]*.

ASCII	*Hex*	*Representation*
0	0x0030	interpret remaining XX bits as an *ItemIdentifier*
i	0x0069	signed integer
u	0x0075	unsigned integer
f	0x0066	IEEE float
c	0x0063	ASCII character
b	0x0062	boolean

The unpack character '0' allows the unpack string to reference another *Item*, thus allowing hierarchical constructs. Where the *bit_length* (or *ItemIdentifier*) field has the same size as your SPEAD flavour's ItemIdentifierWidth.

8.3.5 ItemDescriptorShape

Numerical Identifier : 0x0012

Name : descriptor_shape

Description : Binary packed vector explaining the shape of this *Item*, in terms of the number of *ItemTypeItems* per dimension.

Type : binary packed vector

Shape : *variable length binary field with one dimension*

The *ItemShape* string allows multiple data constructs of the type specified in the *ItemType* unpack string to be placed contiguously in the data stream. These values are binary packed integer values. It has the following form:

```
count_axis1[,count_axis2[,count_axis3 [...]]]
```

where each count_axis field is *HeapAddrWidth*+1 bytes long. The LSb of the first byte of this field indicates if the value is given immediately or otherwise references another item, i.e. it is possible to have variable-length arrays. This size allows indexing of a 1 bit vector along the entire heap.

8.3.6 ItemDescriptorDType

Numerical Identifier : 0x0015

Name : descriptor_dtype

Description : String representation of numpy dtype.

Type : string

Shape : *variable length, single dimension*

The optional descriptor dtype allows specification of the type and shape of an *Item* using a numpy style dtype string. If this string is present the specified shape and type fields, if any, are ignored and populated from the decoded numpy dtype.

The current reference implementation makes use of this dtype specifier to unpack items directly to numpy arrays, realising a significant speedup. Thus, the advantage of this parameter may depend on the specific receiver implementation in use.

8.3.7 Example *ItemDescriptor*

For example, if the user wished to pack a 100x100 pixel RGB-24 image consisting of 8 bit numbers, you might use the following *ItemDescriptor* (as illustrated in Figure 4).

descriptor_identifier : 0x5555
Assign numerical identification code as an unsigned integer with immediate addressing.

descriptor_name : "my_picture"
Short string of length 10 characters interpreted as variable name.

descriptor_description : "A 100x100 pixel RGB-24 image consisting of structs of unsigned 8b red, 8b green and 8b blue data. Data is packed in rows (i.e. you get 100 pixels from row-1, followed by 100 pixels from row-2 etc)."
A verbose, human readable string of length 195 characters.

descriptor_type : u8u8u8
ItemDescriptorType represents the data type (1 byte) and number of bits per component (with bitwidth defined by the ItemIdentifierWidth of 24 bits). Thus, the number of bytes required to represent the type description: 3(1+24/8) = 12 bytes.

descriptor_shape : 100,100
ItemDescriptorShape is read as a sequence of unsigned integers representing the variable size along each dimension. The number of bytes required to represent the shape description is defined by the HeapAddressWidth (40 bits) + 1 byte: 2(1+40/8) = 12 bytes. For this example the LSb of the first byte of each component is zero (0), indicating that dimensions value follows immediately. (If the LSb of the first byte is one (1), it indicates that a pointer to another *Item* follows).

The type and shape fields shown above could be replaced by a numpy dtype string:

descriptor_dtype: "'descr':'u1', 'fortran_order':False, 'shape':(100,100,.3),"

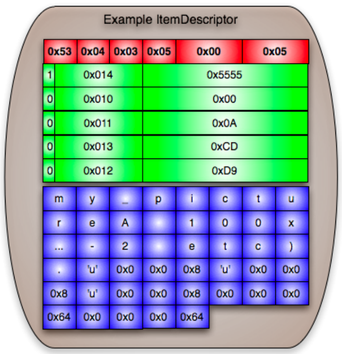

Figure 4: Example *ItemDescriptor* constructed from metadata provided above.

9 Extra features

9.1 HEAP processing

Normally, the system waits for the first packet from the next *Heap* to arrive before beginning processing of the currently received *Heap*. This feature is useful when heap sizes need to be flexible. This is typical when the receiver does not know when it has received all the data for a *Heap*.

However, if the receiver knows the size of the heap beforehand, it can start unpacking as soon as all data has arrived. This allows the protocol to deal with systems which needs to transmit a response to an incoming data stream before receiving the next piece of data (see *Item* ID 0x0002).

It is also possible to have multiple *Heap*s open at the same time. This is due to network reordering of UDP packets, the system can have packets from multiple *Heap*s being received at the same time. In this case you do not want to start processing the previous *Heap* until you are satisfied that all packets have arrived.

Buffer "windows" allow temporary storage of incoming packets associated with up to (N-1) *Heap*s. The system will wait for the first packet of the Nth *Heap* to begin processing of the the *Heap* with the lowest heap counter (see *Item* ID 0x0001).

In general, it is highly recommended that the user transmits the heap size (*Item* ID 0x0002) wherever possible.

A Example Packet Exchange

To illustrate the process of encoding and assigning new *Item*s we consider an example where we want to transmit the 3 variables described in Figure 1 found in the Data Overview Section. These variables are a string, MyString, an integer counter, MyCntr, and an array of 32-bit unsigned integers, MyArray.

1. For each new variable an *ItemDescriptor* is generated. This is necessary for the receiving system to know how to interpret and unpack the data. These packets must be sent before data transmission begins.

2. *ItemDescriptor*s are issued once, at the start of the stream and then only need to be re-sent if parameter shapes change. Full *ItemDescriptor* packets can also be regenerated upon request through out-of-band signalling, allowing a receiver to process a stream which had been started earlier.

   When a data stream terminates, a further packet may be sent. This may contain a stream control *Item* to indicate that the transmission has completed.

3. During transmission of the data stream, packets should contain only the changed *Item* fields, in order to maximise the utilisation of the link. The receiver keeps previous state of *Item*s. Thus, it is only when the value of an *Item* changes that it need to be retransmitted.

4. Received *ItemPointer*s are added to the associated *Heap*. The immediate-addressed *Item* with identifier 0x167's (MyCntr) value is available immediately. *Item* 0x168 (MyArray) must wait for the *Heap* unpack before processing. Packet 1's payload is inserted into *Heap* 1 at offset 0x0. Packet 2 is still pending and the data from *Item* 0x168 is only partially available.

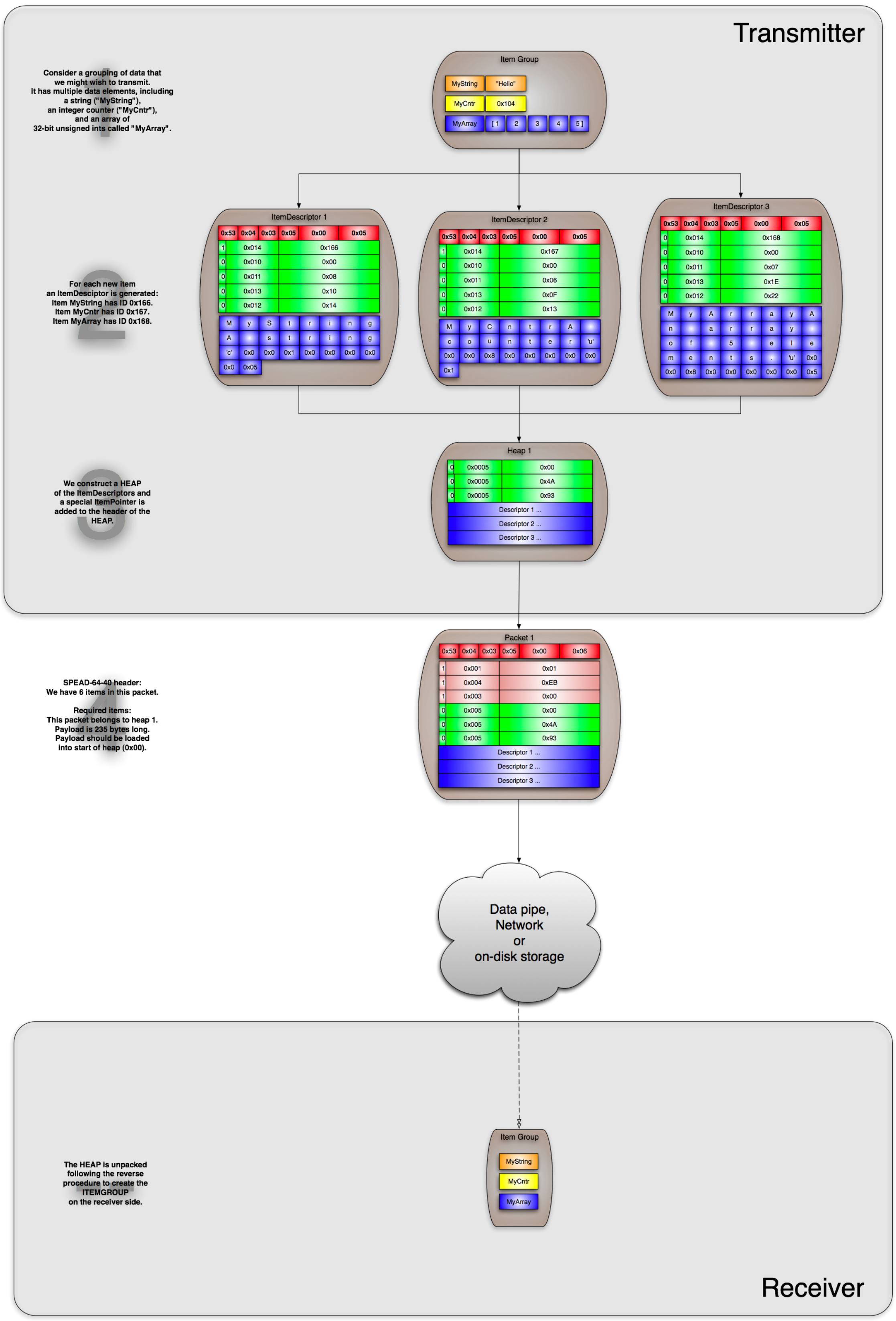
Transmitter
Consider a grouping of data that we might wish to transmit. It has multiple data elements, including a string ("MyString"), an integer counter ("MyCntr"), and an array of 32-bit unsigned ints called "MyArray".
1
Item Group
MyString "Hello"
MyCntr 0x104
MyArray [1 2 3 4 5]
For each new item an ItemDesciptor is generated: Item MyString has ID 0x166. Item MyCntr has ID 0x167. Item MyArray has ID 0x168.
2
ItemDescriptor 1
0x53 0x04 0x03 0x05 0x00 0x05
1 0x014 0x166
0 0x010 0x00
0 0x011 0x08
0 0x013 0x10
0 0x012 0x14
M y S t r i n g
A s t r i n g
'c' 0x0 0x0 0x1 0x0 0x0 0x0 0x0
0x0 0x05
ItemDescriptor 2
0x53 0x04 0x03 0x05 0x00 0x05
1 0x014 0x167
0 0x010 0x00
0 0x011 0x06
0 0x013 0x0F
0 0x012 0x13
M y C n t r A
c o u n t e r 'u'
0x0 0x0 0x8 0x0 0x0 0x0 0x0 0x0
0x1
ItemDescriptor 3
0x53 0x04 0x03 0x05 0x00 0x05
0 0x014 0x168
0 0x010 0x00
0 0x011 0x07
0 0x013 0x1E
0 0x012 0x22
M y A r r a y A
n a r r a y
o f 5 e l e
m e n t s . 'u' 0x0
0x0 0x8 0x0 0x0 0x0 0x0 0x0 0x5
We construct a HEAP of the ItemDescriptors and a special ItemPointer is added to the header of the HEAP.
3
Heap 1
0 0x0005 0x00
0 0x0005 0x4A
0 0x0005 0x93
Descriptor 1 ...
Descriptor 2 ...
Descriptor 3 ...
SPEAD-64-40 header: We have 6 items in this packet.
Required items: This packet belongs to heap 1. Payload is 235 bytes long. Payload should be loaded into start of heap (0x00).
4
Packet 1
0x53 0x04 0x03 0x05 0x00 0x06
1 0x001 0x01
1 0x004 0xEB
1 0x003 0x00
0 0x005 0x00
0 0x005 0x4A
0 0x005 0x93
Descriptor 1 ...
Descriptor 2 ...
Descriptor 3 ...
Data pipe, Network or on-disk storage
The HEAP is unpacked following the reverse procedure to create the ITEMGROUP on the receiver side.
Item Group
MyString
MyCntr
MyArray
Receiver

Figure 5: Creating *ItemDescriptor*s to define 3 variables for transmission using SPEAD-64-40.

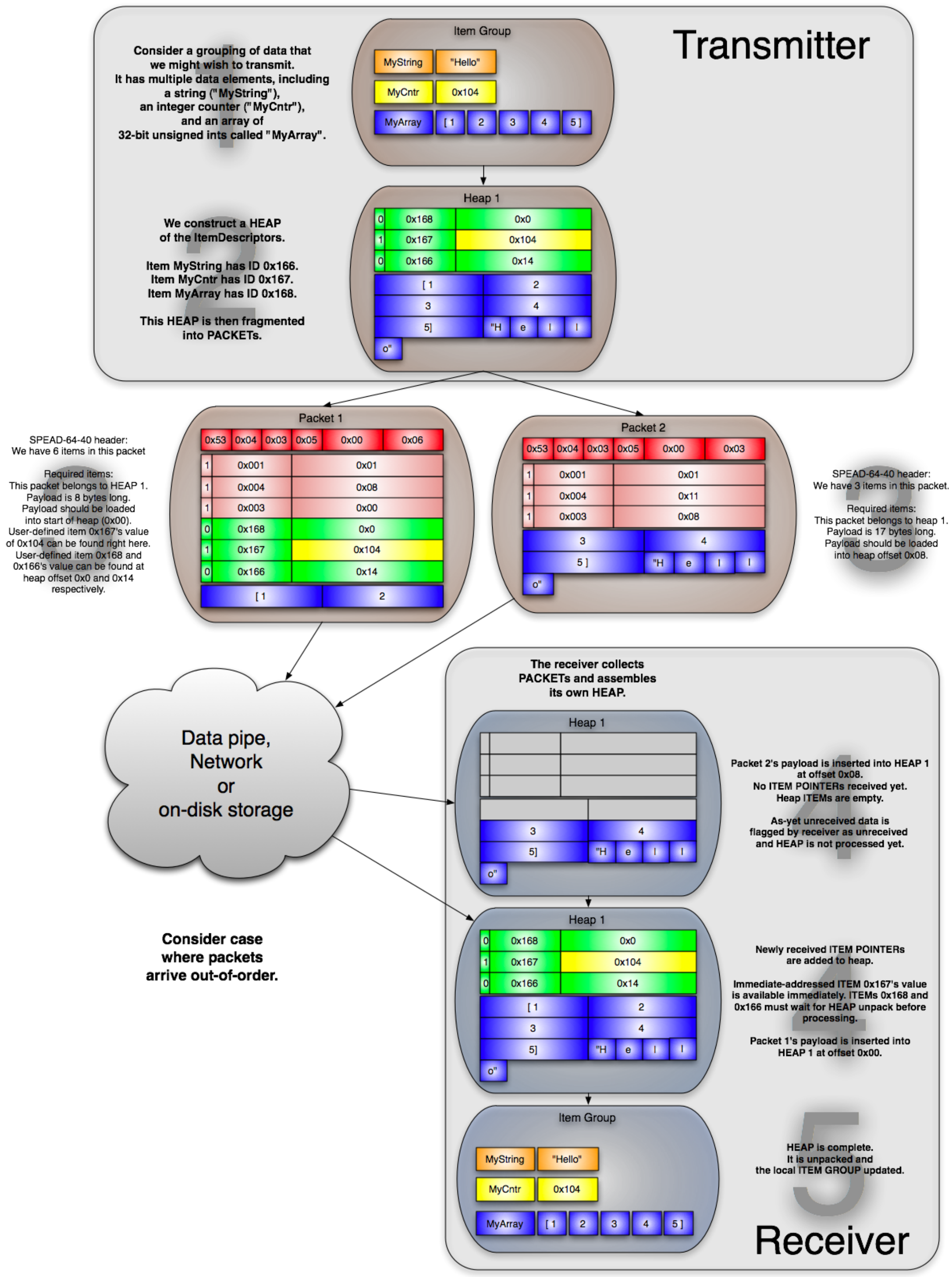
Transmitter
Consider a grouping of data that we might wish to transmit. It has multiple data elements, including a string ("MyString"), an integer counter ("MyCntr"), and an array of 32-bit unsigned ints called "MyArray".
Item Group
MyString "Hello"
MyCntr 0x104
MyArray [1 2 3 4 5]
We construct a HEAP of the ItemDescriptors.
Item MyString has ID 0x166.
Item MyCntr has ID 0x167.
Item MyArray has ID 0x168.
This HEAP is then fragmented into PACKETs.
Heap 1
0 0x168 0x0
1 0x167 0x104
0 0x166 0x14
[1 2 3 4 5] "H e l l o"
Packet 1
0x53 0x04 0x03 0x05 0x00 0x06
1 0x001 0x01
1 0x004 0x08
1 0x003 0x00
0 0x168 0x0
1 0x167 0x104
0 0x166 0x14
[1 2
SPEAD-64-40 header: We have 6 items in this packet
Required items: This packet belongs to HEAP 1. Payload is 8 bytes long. Payload should be loaded into start of heap (0x00). User-defined item 0x167's value of 0x104 can be found right here. User-defined item 0x168 and 0x166's value can be found at heap offset 0x0 and 0x14 respectively.
Packet 2
0x53 0x04 0x03 0x05 0x00 0x03
1 0x001 0x01
1 0x004 0x11
1 0x003 0x08
3 4 5] "H e l l o"
SPEAD-64-40 header: We have 3 items in this packet.
Required items: This packet belongs to heap 1. Payload is 17 bytes long. Payload should be loaded into heap offset 0x08.
Data pipe, Network or on-disk storage
Consider case where packets arrive out-of-order.
The receiver collects PACKETs and assembles its own HEAP.
Heap 1
Packet 2's payload is inserted into HEAP 1 at offset 0x08. No ITEM POINTERs received yet. Heap ITEMs are empty.
As-yet unreceived data is flagged by receiver as unreceived and HEAP is not processed yet.
Newly received ITEM POINTERs are added to heap.
Immediate-addressed ITEM 0x167's value is available immediately. ITEMs 0x168 and 0x166 must wait for HEAP unpack before processing.
Packet 1's payload is inserted into HEAP 1 at offset 0x00.
Item Group
HEAP is complete. It is unpacked and the local ITEM GROUP updated.
Receiver

Figure 6: Once an *Item* has been described, data packet transmission can begin. In Step 3 of this example we use ridiculously small packets to demonstrate fragmentation.