SDP Memo 046: Experiences with the SPEAD protocol

Document number…… SDP Memo 046
Document Type…….. MEMO
Revision……...1.0
Authors…….. Rodrigo Tobar, Andreas Wicenec, Markus Dolensky, Louisa Quartermaine
Release Date…….. 2018-05-25
Document Classification…….. Unrestricted

Lead Author	Designation	Affiliation
Rodrigo Tobar	Software Engineer	ICRAR
Signature & Date:	Rodrigo Tobar	May 25th, 2018

SDP Memo Disclaimer

The SDP memos are designed to allow the quick recording of investigations and research done by members of the SDP. They are also designed to raise questions about parts of the SDP design or SDP process. The contents of a memo may be the opinion of the author, not the whole of the SDP.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ASKAP: Australian Square Kilometer Array Pathfinder

CSP: Central Signal Processor

OSI: Open Systems Interconnection

MRO: Murchison Radio Observatory

MTU: Maximum Transmission Unit

MWA: Murchison Widefield Array

RTT: Round-Trip Time

SDP: Science Data Processor

SIP: SDP Integrated Prototype

SPEAD: Streaming Protocol for Exchanging Astronomical Data

TCP: Transmission Control Protocol

UDP: User Datagram Protocol

Introduction

The SDP – CSP Visibility Data Interface carries streaming visibility data unidirectionally from the CSP to the SDP instances. It is the primary measurement data interface. The interfaces between SDP and CSP follow the OSI standard and are described in the respective ICDs for SKA1_LOW [RD07] and SKA1_MID [RD08]. Most relevantly, the OSI layers 5 to 7 - the session, presentation and application layers - are implemented by the SPEAD [RD01] protocol. OSI layer 4, the transport layer, uses UDP and OSI layer 3, the network layer, uses IP. The current ICDs specify a maximum allowed correlator output data rate of 0.7 TB/s for Low and 0.6 TB/s for Mid. This corresponds to 80 % SDP occupancy of 72 and 64 x 100 GbE links respectively.

SPEAD currently supports a number of transport mechanism, most notably the User Datagram Protocol (UDP), which needs to be characterized in order to function reliably over a particular link. Therefore, it is important to study how well SPEAD over UDP performs on a link like that connecting CSP and the SDP.

In this memo we show our experience measuring the performance of the SPEAD UDP transport in different networks, including one that resembles very closely the one that will be connecting the SKA1-LOW and the SDP sites. We also include the description of an additional Transmission Control Protocol (TCP) transport implementation developed at ICRAR as an alternative to the default UDP transport. We compare the results of this TCP transports in all environments as well. Most importantly, we compare the speeds achieved with TCP against the data loss rate and speeds achieved by UDP.

This work is part of SDP JIRA ticket TSK-2267 [RD05]. The initial description on the ticket included testing SPEAD stand-alone as well as together with the OSKAR [RD06] streaming interfaces. Given that the complexity of testing SPEAD thoroughly on its own was already high enough, we decided to work on SPEAD investigations directly, leaving the OSKAR part of the tests as future work. Our investigation also provides valuable verification information on the interface between CSP and SDP [RD07], and also assists in identifying and mitigating any associated risk in this area. Our work is complementary and does not appear to be duplicating any SIP work already performed or in progress.

References

Reference Documents

Reference Number	Reference
[RD01]	SPEAD: Streaming Protocol for Exchanging Astronomical Data http://casper.berkeley.edu/astrobaki/images/9/93/SPEADsignedRelease.pdf
[RD02]	spead2 documentation https://spead2.readthedocs.io/en/latest/
[RD03]	original spead2 source code repository https://github.com/ska-sa/spead2
[RD04]	modified spead2 source code repository https://github.com/rtobar/spead2
[RD05]	Experience with the SPEAD protocol https://jira.ska-sdp.org/browse/TSK-2267
[RD06]	OSKAR - SKA Radio Telescope Simulator https://github.com/OxfordSKA/OSKAR
[RD07]	SKA1 LOW SDP - CSP Interface Control Document, 100-000000-002, rev03
[RD08]	SKA1 MID SDP - CSP Interface Control Document 300-000000-002, rev03
[RD09]	Prototype data loss and buffer performance of 40 GbE into multiple 10 GbE links https://jira.ska-sdp.org/browse/TSK-1746

SPEAD design

SPEAD [RD01] is a data exchange protocol. From an application's perspective, SPEAD offers the abstraction of sending or receiving updates on variables (termed *items*) of arbitrary type and shape (e.g., *a two-dimensional array of dimensions NxM of 32-bits integers*). Items are grouped in *item groups*, and a *heap* containing the item group's values is sent over the network. Multiple heaps can be sent through the same *stream*, which represents a communication channel between senders and receivers.

Even though SPEAD already supports a number of stream implementations, its main transport is the *UDP stream*. Other stream implementations include *ibverb* (for efficient InfiniBand communications) and *pcap* (receiver-only, for packet capturing). Even though there exist multiple stream implementations, the SPEAD design assumes UDP as the transport mechanism in some places. The SPEAD description document [RD01] for example details how a heap is first split into *packets*, which are designed to correspond one to one with UDP datagrams. A "start of heap" and "end of heap" mechanism is also specified as part of the packet exchange protocol, which is necessary in UDP streams given its connectionless nature. Finally, packet size is a runtime user parameter, and needs to be set to a value close to the Maximum Transmission Unit (MTU) of the link over which data is sent to achieve best performance. Most notably, even though UDP inspired most of the design, there is no built-in re-transmission mechanism for lost packets, nor a timeout mechanism to give up on receiving a particular heap. The SPEAD receiving software can still deal with incomplete heaps and pass them up to the application, but it is up to the application to implement means to deal with incomplete heaps, which in principle could handle them in such a state. In reality most applications probably want to be handed over fully-received heaps though. Moreover, the SPEAD python bindings only hand over fully-reconstructed heaps to the application.

The SPEAD receiver software design also allows to have more than one *reader* receiving packets that logically belong to the same stream. This way one can receive packets belonging to the same heap from different remote endpoints (e.g., if data is generated and sent in parallel) and aggregate them seamlessly.

UDP stream features

The existing SPEAD UDP stream implementation features include:

- It supports both unicast and multicast communication, enabling one-to-many and many-to-many data transfers, which would be cumbersome and costly using other transports.
- It benefits from the fact that UDP transmission rates do not depend in principle on the delay of the link over which data is sent since there is no response sent.
- Senders can be throttled to a specific rate. This option is made visible to applications so they have full control on the sending speed.

- Because no re-transmission is supported by SPEAD, data can be lost. This can be totally acceptable (up to some point) by some applications, but not by others. Because the loss rate over a specific link depends mostly (but not only) on the sending rate, careful benchmarking needs to be performed to find the speed that yields an acceptable loss rate.
- Related to the previous point, and as a consequence of the serialization of heaps into packets, the granularity of data loss is increased as seen by the application, since even a single packet being lost means that the whole heap is lost.
- A second consequence of the data loss is that losing end-of-heap packets in particular can lead to hanging clients. In most cases the lack of a timeout will not manifest itself because higher-level ring-buffers (offered by the SPEAD API) ensure incomplete heaps are discarded as more data arrives. There are situations however where a stream is meant to be shut down after receiving a finite number of heaps (e.g., during short-lived interactions, or during a benchmarking situation), and the packet signaling the end-of-heap for the last heap is missed, causing the stream to hang waiting for the missing packet to arrive.

A TCP stream implementation for SPEAD

As part of our experience we added support for TCP transport to the SPEAD protocol. Using TCP has inherent advantages, like built-in reception acknowledgement, retransmission, timeout and congestion control handling, all of which can be desirable properties in many scenarios. It also comes with downsides, mainly its dependency on the round-trip time (RTT) and proper buffer sizing to saturate network links.

The SPEAD code [RD03], written in C++11, is very well documented and designed. It heavily uses the *Boost.Asio* (Asynchronous I/O) library, which encapsulates and abstracts all asynchronous networking operations. Given these conditions, adding an initial, non-intrusive sender and receiver TCP implementation that followed the current SPEAD design was fairly simple. On the sender side a new `spead2::send::tcp_stream` class (similar to `spead2::send::udp_stream`) that uses TCP sockets instead of UDP sockets to send the SPEAD packets was created. On the receiver side, a corresponding new `spead2::recv::tcp_reader` class (similar to `spead2::recv::udp_reader`) was created to receive the packets via a TCP socket. In both cases most of the logic is already abstracted in base classes, making the new code quite compact. Finally, we adapted the `spead2_send` and `spead2_recv` programs to optionally use the TCP sender and receiver classes respectively. Integration with the rest of the spead2 software, including the benchmark tools and the python bindings, is future work.

In terms of protocol, the TCP connects (via `connect(2)`) to the TCP receiver (which listens via `listen(2)`). All the data is then sent by the sender, who finally closes the connection. Upon sender disconnection the receiver also disconnects and the TCP connection is then fully closed.

In principle more than one TCP reader could be used to receive packets from different remote endpoints into a single stream (see SPEAD design), and therefore reconstruct heaps created in parallel, like in the case of CSP (as described in [RD07]). The `spead2_recv` tool supports this topology (i.e., it allows more than one reader to be attached to a stream) but the `spead2_send` tool does not, regardless of the underlying transport. Thus we were not able to corroborate that this functionality works.

The main difference between the existing UDP transport and our TCP transport implementation is how SPEAD packets are treated. The SPEAD packets are designed with UDP in mind: once a heap is fragmented, it is very easy to send its packets over UDP, and to receive them and reassemble them on the receiving side. Given the nature of TCP (which presents the data as a stream of bytes rather than a series of messages), this fragmentation into SPEAD packets does not map well to how data is transported. First, the sender needs to notify the receiver of the size of each SPEAD packet prior to sending it. Then the receiver needs to read the packet size, read the data, and then reconstruct the packet. Coding and decoding SPEAD packets via TCP adds computational overhead and marginally increases the data volume (by less than 0.1% in our experiments). The number of extra bytes at the moment are also not reported correctly by the TCP sender class, but then again the amount is so small that it does not change the results.

The effects described above are specific to this first implementation and are due to keeping changes to the code at a minimum. An alternative, more efficient TCP transport implementation would not fragment a heap into packets on the sender side, sending the contents of the heap straight out to the receiver, without having to reassemble packets (i.e., modifying the heap being reconstructed directly instead of creating a packet and adding it to the heap). This would reduce computational loads, both on the sender and receiver sides, and the amount of data transmitted on the wire. Again, these code changes were outside the scope of this exercise.

SPEAD over low-latency links

To get a more detailed understanding of the behavior of each transport, we tested the UDP and TCP transports (both over IPv4) against two different low-latency links:

- Loopback: This is the *lo* interface in a Linux machine. The particular machine used for these tests (hostname *bolano*) features an Intel Core i7-5600 CPU @ 2.6 GHz, and a Linux 4.13 kernel in an Ubuntu 17.10 installation. The measured RTT after pinging with 30 packets is 0.061 ± 0.022 [ms]. The MTU is of 65536 bytes.
- 1G: This is a 1 [Gb/s] ethernet link between *bolano* and one of the machines in our local cluster (hostname *sorrento*). The link requires hopping through two switches before reaching the destination. The measured RTT after pinging with 30 packets is 0.298 ± 0.069 [ms]. The MTU is 1500 bytes.

Before running any SPEAD benchmarks we tested both links using *iperf* to measure their maximum data exchange rates. Results are shown in table 1. Each experiment was ran 5

times, each with a duration of 20 seconds, and used the highest possible value for packets/buffers being sent to avoid IP fragmentation[1] without changing any kernel settings.

Link	UDP/IPv4 (speed, loss rate)	TCP/IPv4 (speed)	Comments
Loopback	56.7 ± 0.46 [Gb/s] 0.26 ± 0.22 [%]	51.82 ± 1.01 [Gb/s]	
1G	810.20 ± 0.40 [Mb/s] 99.94 ± 0.00 [%]	935 ± 0.00 [Mb/s]	Even with sending rates of ~50 [Mbits/s] we got loss rates of ~99 [%]

Table 1: Raw link performance using different protocols. UDP bandwidth was set differently depending on the maximum bandwidth of the link.

These results give a maximum bandwidth value for each of the links, which allows us to compare the performance of the SPEAD sender/receive stacks and calculate their overheads. In particular, the loopback interface allows for 50+ [Gb/s] speeds both for TCP streams and UDP streams, with UDP transmissions (at these speeds) dropping a small number of packets, and being only ~10% faster than TCP (which doesn't drop any packets).

We then tested the *spead2_send* program sending data through the loopback interface against two different receivers: first, a *light* receiver (*iperf* for TCP, none for UDP) to observe the performance of the sender in the presence of a quick receiver, and have an idea how fast the software stack itself runs. The second receiver used during the tests was the *spead2_recv* program, which allows us to understand how well the two stacks work together, including transmission speed and packet loss ratio (in the case of UDP). Finally, for each combination of protocol and receiver, we tested four different configurations:

- *Default*. This is the out-of-the-box experience. No tweaking is done neither on the sender nor in the receiver side. No kernel parameters have been changed
- *MTU*. Like *default*, but configuring the sender and the receiver to use bigger SPEAD packet sizes, as close to the MTU of the link as possible.
- *Buffers*. Like *default*, but with both the kernel maximum read/write network buffers set up to 16 [MB] and the application's socket buffers set to the same amount.
- *MTU* + *buffers*. Including both *MTU* and *buffers* configuration changes.

Light receiver

	Loopback speeds against *light* receiver [Gb/s]			
Proto	Default	MTU	Buffers	MTU + buffers

[1] This was the MTU of the link minus the size of the IPv4 (20) and UDP (8) headers in the case of UDP, and the MSS reported by iperf in the case of TCP.

UDP	2.28 ± 0.02	43.13 ± 0.26	2.19 ± 0.08	42.83 ± 0.90
TCP	3.09 ± 0.06	27.91 ± 0.55	3.14 ± 0.04	25.96 ± 1.62

Table 2: Sending speeds when running the *spead2_send* program against a light receiver.

Table 2 shows the sending speed of the SPEAD sender program for the two transports under different configurations when running over the *loopback* interface. Under the default configuration both transports clearly underperform, but increasing the SPEAD packet size close to the MTU size increases the speeds noticeably in both cases, while buffer sizes don't affect the experiments, as expected (in the case of UDP we are not measuring loss rates, and in the case of TCP the round-trip is too low for the buffer sizes to make much difference). In their best cases the speed of both protocols are well under the network bandwidth limits though (as seen in the previous subsection), and therefore speeds in Table 2 can be seen as the upper bandwidth limit at which the SPEAD sender software stack can work. In particular, it can be noted that SPEAD UDP senders can run up to ~43 [Gb/s], while SPEAD TCP senders can run up to ~28 [Gb/s].

SPEAD receiver

Loopback interface

	Speeds against *spead2_recv* [Gb/s] Heap loss rates [% of heaps]			
Proto	Default	MTU	Buffers	MTU + buffers
UDP	**2.61 ± 0.12** 2.73 ± 0.87	**37.10 ± 0.23** 99.38 ± 0.28	**2.51 ± 0.11** 0.00 ± 0.00	**24.55 ± 0.22** 99.80 ± 0.18
TCP	**3.50 ± 0.05**	**28.82 ± 0.21**	**3.58 ± 0.03**	**15.10 ± 0.06**

Table 3: Sending speeds and heap loss rates when running the *spead2_send* program against the *spead2_recv* program.

Table 3 shows the results of running the SPEAD sender program against the corresponding SPEAD receiving program over the loopback interface, under different configurations, and using the different underlying protocols. The results are mostly similar to those against a *light receiver*, except from some differences. Again, increasing the SPEAD packet size close to the MTU size increases the speeds noticeably. Note however how the UDP protocol misses a big number of heaps from the sender, even though the initial iperf tests showed a low datagram loss rate. This is probably due to: a) the fact that the SPEAD receiver adds an additional overhead due to the required deserialization of SPEAD packets into heaps, and b) the fact that a SPEAD heap is lost if even if just one of the contained packets is lost, effectively increasing the granularity of the loss. On the other hand the TCP transport achieved almost 30 [GB/s] of speed without loss. This shows that both the sender *and* receiver SPEAD TCP software stack can run at these speeds, and that any other

bottlenecks will be non-CPU related. Increasing the buffer sizes for this particular link was of no use, since the latency is so low, and it effectively lowered the data transfer rates both for UDP and TCP.

When comparing these results to those shown in Light receiver, it is interesting to note how adding a UDP receiver took down the maximum speed of the sender/receiver pair to ~37 [Gb/s] (from the original ~43 [Gb/s]), while the TCP sender/receiver pair remained at the original ~28 [Gb/s] working speed. This result shows that (single-threaded) TCP receivers can work at such speeds as well, and because TCP receivers share most of the code with UDP receivers the same statement is most probably true for lossless UDP receivers.

1 [Gb/s] interface

Finally, we test the same SPEAD sender/receiver pair over a 1 [Gb/s] link.

	Speeds against *spead2_recv* [Gb/s] Heap loss rates [% of heaps]			
Proto	Default	MTU	Buffers	MTU + buffers
UDP	**0.95 ± 0.00** 1.60 ± 0.65	-	**0.95 ± 0.00** 0.00 ± 0.00	-
TCP	**0.93 ± 0.00**	-	**0.93 ± 0.00**	-

Table 4: Sending speeds when running the *spead2_send* program against the *spead2_recv* program via the 1G link connecting *bolano* and *sorrento*.

Table 4 shows the results of sending data from the SPEAD sender to the SPEAD receiver via the 1G link. The results show how UDP is faster than TCP (although only by ~2%), but still misses packages in the default configuration. Changing the kernel buffer maximum limit to a suitable value for the bandwidth of the link results in a completely lossless transmission for UDP. Since it is clear that using packets of size close to the MTU is the best we only tested that. Moreover, the SPEAD programs already default to using packets of a size that is safe for a standard IPv4 over Ethernet setup.

SPEAD over an SKA1-like link

As part of our evaluation of SPEAD we also tested the sender and receiver software over a fibre link connecting the Murchison Radio Observatory (MRO) with a Perth-based set of machines. The MRO currently houses both the ASKAP and MWA instruments, and its network distance to Perth is about 890 km. Because the SKA-LOW is planned to be located next to the MRO the distance covered by this link is effectively the same that packets will have to travel from the CSP correlator to the SDP system. The link is currently not used by

any system, and it is part of the MWA expansion to its Phase 2, and completely separate from the link transporting data for MWA Phase 1.

An overview of the available hardware for testing (shown in Figure 1) is as follows:

- At MRO, a machine (hostname *vcs17*) with two 40 [Gb/s] interfaces is connected to a Cisco Nexus 9000-series switch.
- A second machine at MRO (hostname *medconv*) with one 10 [Gb/s] interface is connected to the same switch above. *medconv* actually has two such interfaces, but one of them was not in working order so could not be used for these experiments.
- In Perth, ten machines (hostnames *mwacache01* to *mwacache10*) with two 10 [Gb/s] interfaces each are connected to a second Cisco Nexus 9000-series switch.
- Both switches are directly connected through a fibre link running at 100 [Gb/s].

Incidentally, this setup seems very similar to that described in [RD09], although probably a more thorough testing was conceived for [RD09].

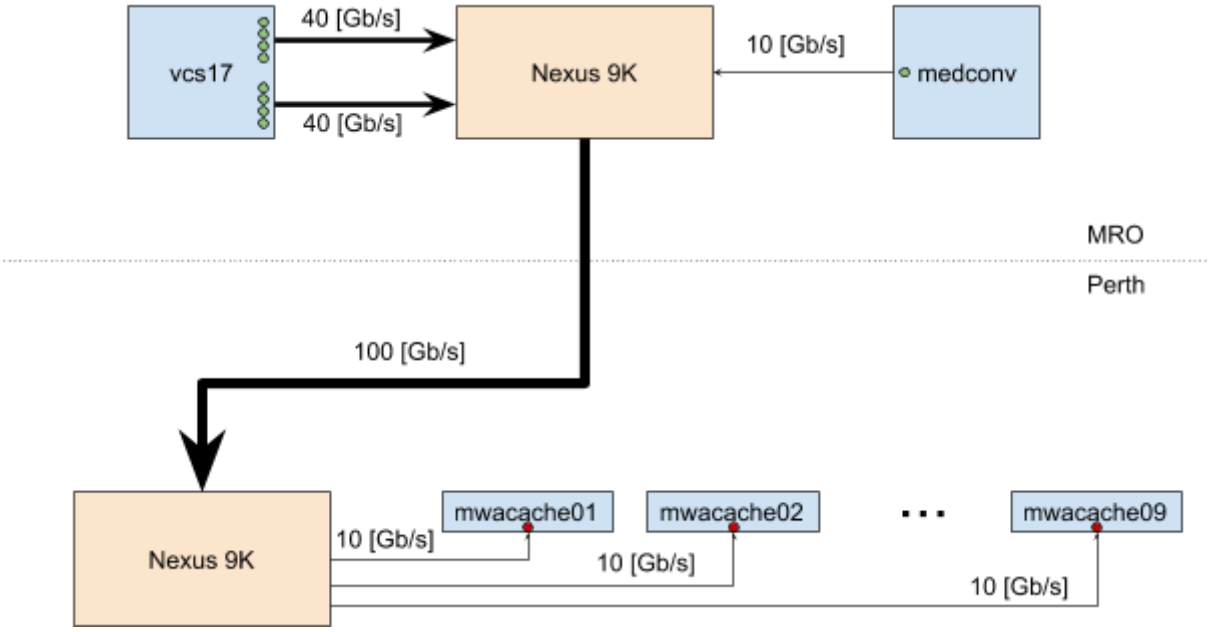

Figure 1: 100 [Gb/s] link test setup. Computers are blue, switches orange. Green dots are 10 [Gb/s] senders, red dots are receivers. The MRO and Perth sites are separated by 890 [km]

To use most of the bandwidth available on the 100 [GB/s] link we use the two 40 [Gb/s] interfaces on *vcs17* and the working 10 [Gb/s] interface on *medconv* to saturate the link from the sender side, which should give us a theoretical maximum sending capacity of 90 [Gb/s]. On the receiving side we use nine of the ten *mwacache* machines to receive incoming data on one of their 10 [Gb/s] interfaces. We start one receiver on each of the *mwacache* machines. Because we are limited by our receiving speed we start senders bound to no more than 10 [Gb/s] each. One sender is started on *medconv* and eight on *vcs17* (four bound to each interface). In other words, nine individual 10 [Gb/s] connections are made between the MRO and Perth.

The measured RTT of the end-to-end link after pinging with 30 packets is 8.77 ± 0.128 [ms], as measured between *vcs17* and *mwacache01* (and extremely similar when measured from *medconv*). Using this measurement, and based on the fact that even on *vcs17* we will use only up to 10 [Gb/s] per connection, we set the network maximum reading or writing (as corresponding) buffer sizes in all machines to 12 [MB]. Jumbo packets are enabled across the entire system, allowing communications with an MTU of 9000 bytes. Other than that, the switch configuration remained unchanged.

Experiment details

In the following subsections we show three experiments conducted on the MRO Precursor Network:

1. Single 10 [Gb/s] sender on each available sending interface [MRO10]
2. Multiple senders on a single 40 [Gb/s] interface [MRO40]
3. 9 x 10 [Gb/s] sender/receiver pairs [MRO90]

During each experiment we send 50000 heaps with one item each through each individual 10 [Gb/s] stream being setup. SPEAD packet sizes were set in all cases to 8950 bytes to closely match the 9000 MTU size of the link. The heap payload size was set to 4 [MB], the default setting on the sender program, therefore yielding ~470 packets per heap (the precise number depends on the amount of metadata SPEAD adds to each packet). With that payload size, 50000 heaps translate to almost 200 [GB] of payload data. Given the capacity of the link this means that even the shortest test measurement ran for more than 3 minutes, and that during the full deployment tests we sent close to 1.8 [TB] of data (450000 heaps) through the link. For UDP streams multiple sending speeds were measured using the built-in speed rate throttling mechanism available in SPEAD to understand the correlation between UDP speeds and error rates. TCP senders were left unbound to the maximum speed they could achieve. UDP and TCP speeds were always measured on the sending side of the stream, and heap loss rate on the receiver side. Aggregated speeds were calculated by dividing the total number of bytes sent by the total amount of time it took to finish all sender processes (rather than simply summing the speeds reported by the senders). The aggregated heap loss rate during UDP transmissions was calculated as the total number of heaps received over how many should have been received.

It is important to note that in some cases when sending data through UDP the precise number of received heaps was not available. This was because the end-of-heap SPEAD packet was not received, which leaves the receiver (as currently implemented) in a hanging state waiting for such packets, as discussed in UDP stream features. A timeout mechanism could be implemented on top of the current UDP receiver logic to detect a stalled stream and finish the receiving of data.

Configuration MRO10: Single per-interface tests

To have a first idea of how well the system performs we first measure the performance of only three connections: two using each of *vcs17*'s interfaces and a third using *medconv*'s interface. This gives us an idea of how well *vcs17*'s interfaces perform when sending a single 10 [Gb/s] stream of data, and tell us already how well the receiving end performs. Note that the theoretical upper bound for the transmission rate in this experiment is 30 [Gb/s].

Results for this test are shown in Figure 2. In this configuration the maximum reasonable per-connection UDP speeds that can be achieved are between 9.5 [Gb/s], with a heap loss rate of 0.4%, and 9.6 [Gb/s] with a heap loss rate of 1.75%. These speeds yield aggregate speeds of ~28.3 [Gb/s] and ~28.7 [Gb/s], respectively, representing a utilization of the link of ~95%. The UDP error rate rises dramatically when exceeding 95 % bandwidth saturation. It is compounded by the fact that one missing packet corrupts the complete heap. Increasing bandwidth saturation from 95 to 97% increases the error rate from 0.4 to 40%. Finally, aggregated UDP speeds scale linearly with the individual UDP rates, which is expected as the three streams map to three different interfaces.

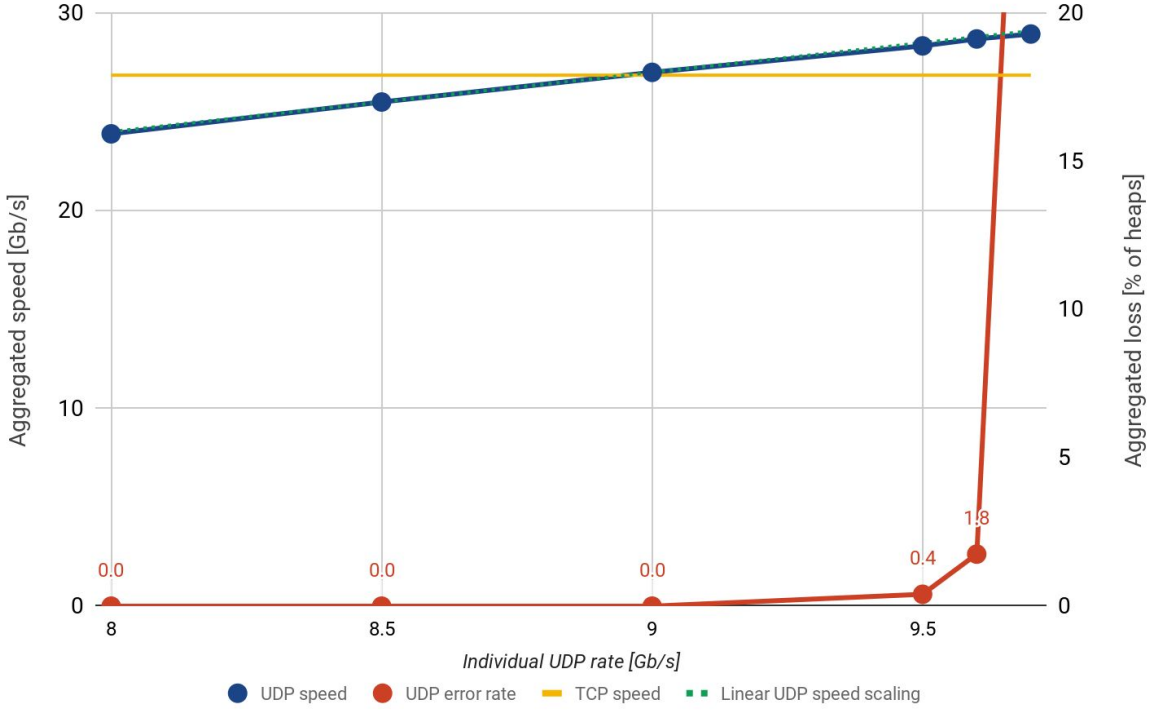

Figure 2: Performance of three parallel 10 [Gb/s] connections deployed in three different sending interfaces. Dark blue points are aggregated UDP speeds, red dots are UDP error rates, the dotted green line is the linear scaling of the individual UDP rates, and the yellow line is the aggregated TCP speed. UDP aggregated speeds scaled almost perfectly, while the aggregated TCP speed agrees almost perfectly with UDP speeds presenting no data loss. After 9.5 [Gb/s] the error rate goes up quickly, with values of 1.8% at 9.6 [G/bs] and 38.3% (outside of the plot) at 9.7 [Gb/s].

In the same scenario the TCP-based connections yield an aggregate speed of ~26.9 [Gb/s], representing a utilization of almost 90% of the link. This result corresponds quite well with the performance of lossless UDP.

Given these results, it can be said that receivers are working in good condition, that both switches are doing a good job at multiplexing data streams coming from different machines, and that both UDP and TCP transports offer good link usage.

Configuration MRO40: Multiple connections, single interface tests

Having studied how a single stream coming from each of the sending interfaces behave over the link, we now test how well multiple streams being sent from a single interface in *vcs17* perform. This experiment will show, in isolation, how well we can use each of these interfaces to effectively saturate the MRO-Perth link. This is done by starting four senders on *vcs17* that will use the same interface to send their data, and their respective receivers on the *mwacache* machines.

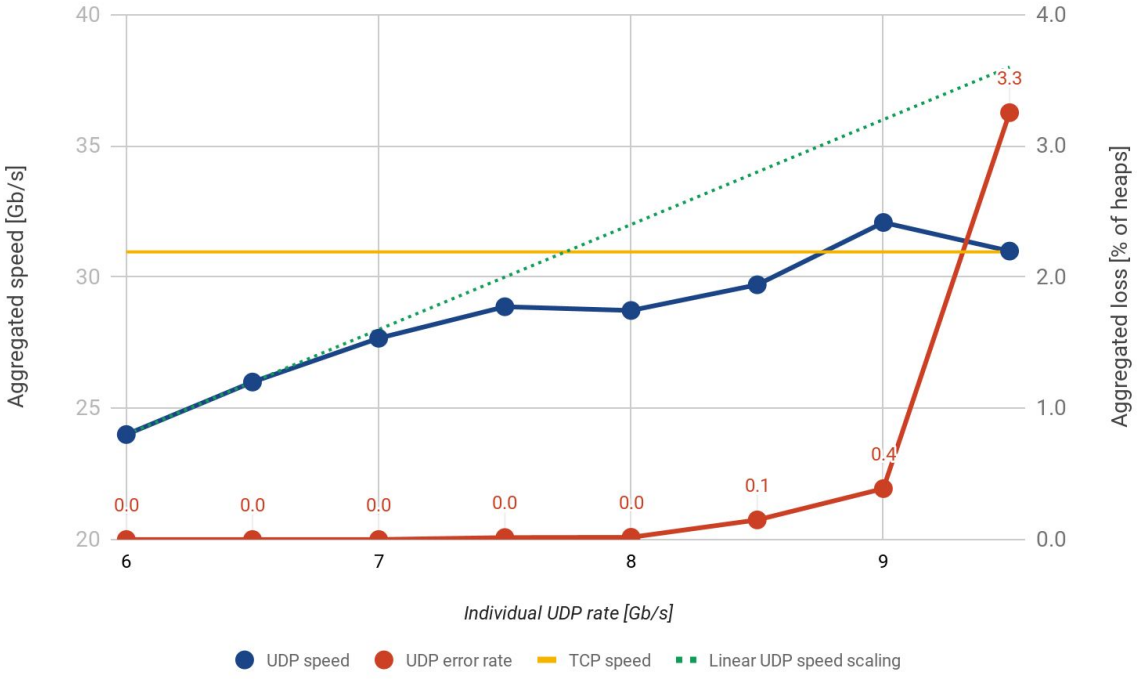

Figure 3: Performance of four parallel 10 [Gb/s] streams being sent through the same 40 [Gb/s] interface. Dark blue points are aggregated UDP speeds, light blue dots are UDP error rates, the green dotted green line is the linear scaling of the individual UDP rates, and the yellow line is the aggregated TCP speed. UDP aggregated speeds stop scaling and seem to saturate at around 32 [Gb/s]. Aggregated UDP speeds with <1% data loss rates again agree with the aggregated TCP speed.

Results for this experiment are shown in Figure 3. Like in the previous experiment, different UDP sending rates are sampled, but only up to 9.5 [Gb/s] in this case since we know from the previous experiment that individual streams already saturate at higher speeds. The first thing to note is how the aggregated UDP sending rate stops scaling after a certain point, flattening around the 32 [Gb/s] mark, representing an ~80% of the actual interface speed. This indicates a bottleneck somewhere else in the system , which we investigate further below. The aggregated UDP speed after which loss rates become greater than 1% is also

around 32 [Gb/s], which is very similar to that shown by the ~31 [Gb/s] speed achieved by the TCP backend.

Regarding the bottleneck observed in the system, we know from Light receiver that SPEAD UDP senders can run to speeds of up to ~43 [Gb/s]. We repeated the experiment on *vcs17*, and confirmed that an rate-unbound UDP sender can send data at ~45 [Gb/s] through the loopback interface, therefore discarding the CPU as the bottleneck. We then sent data through one, two, three and four UDP senders through one of the 40 [Gb/s] interfaces at different individual speeds.

Results of these measurements can be found on Figure 4. Points represent measured aggregate UDP speeds for different number of parallel senders (from one to four), and lines show theoretical linear scales. Points and lines of the same color correspond to the same data rate for each individual UDP sender, with measurements ranging from 9 to 20 [Gb/s]. These results are consistent with the behavior we observed during the initial experiment depicted in Figure 3, where the maximum sending rate we targeted was using four parallel senders at 9.5 [Gb/s] each. In that case we observed an aggregate sending rate of around 31 [Gb/s], while in these new results we see aggregate speeds of around 32 [Gb/s] for four senders running both 9 and 10 [Gb/s] individual speeds. This therefore represents the maximum achievable speed we can expect when using *vcs17*'s interfaces, since all our experiments are constrained by the receiving speed of 10 [Gb/s].

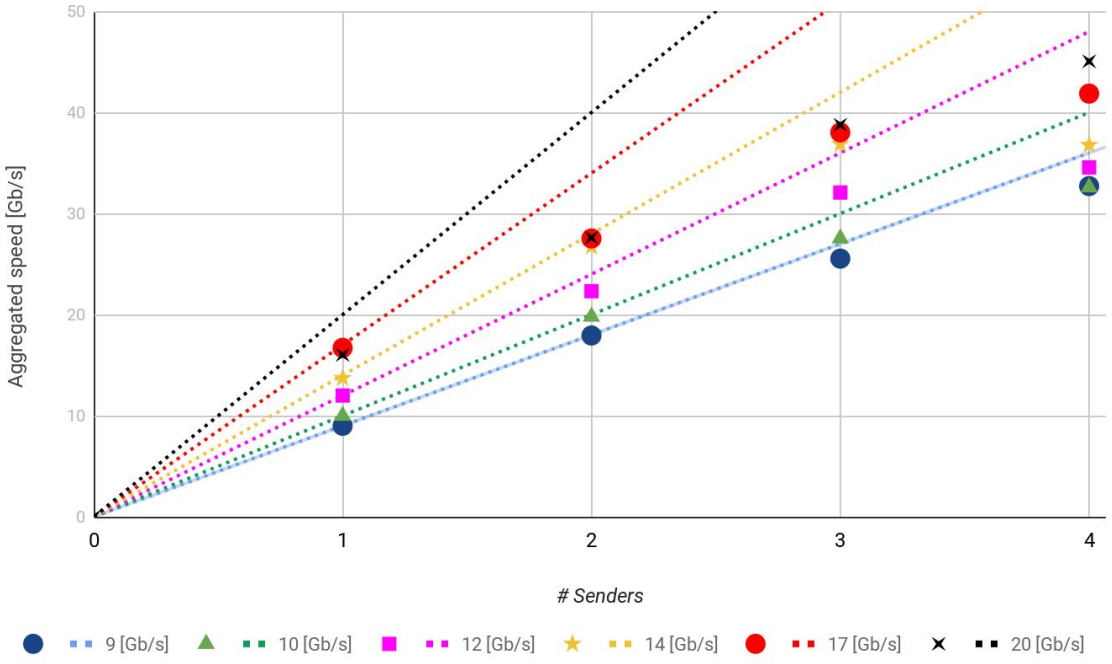

Figure 4: Scaling of different number of UDP senders at individual speeds of 9, 10, 12, 14, 17 and 20 [Gb/s]. Points show measured values, while lines sharing the same color show the theoretical linear scaling. Scaling deterioration can be observed both at high aggregated rates and at high individual rates.

Figure 4 also shows two main factors contributing to the scaling of the aggregate sending speed. The first is the fact that individual senders seem unable to send data at more than ~17 [Gb/s]. This is visible most notably with the measurements for 20 [Gb/s] senders (red points and lines), where even with one sender only ~17 [Gb/s] are reached. The second factor is the target aggregated speed itself, with higher target data sending rates producing increasingly worse scaling results. This effect, depicted in Figure 5, seems to be independent of the individual rates or the number of senders by themselves. It is still unclear what is producing this bottleneck in the system, and further investigation will need to be done to find the exact cause. Examples of possible causes are PCIe congestion, memory bandwidth limitations, or other bus speeds constraining the system.

Finally, it can be noted that when using four parallel senders aggregated speeds over 40 [Gb/s] were reached. This is most probably due to the kernel dropping datagrams even before placing them in the network interface's own buffer due to the high writing load.

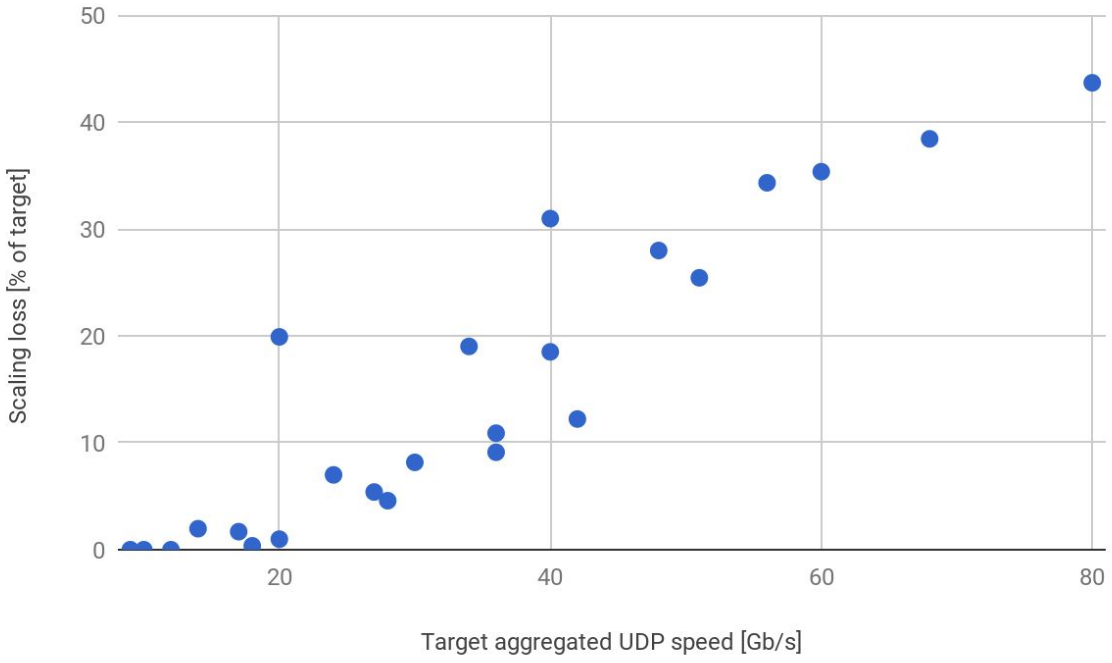

Figure 5: Scaling loss as a function of the target aggregated UDP speed. This is the same data shown in Figure 4. The scaling loss is the difference (in percent) between each measurement (points in Figure 4) and its corresponding theoretical limit (lines in Figure 4).

Configuration MRO90: 9 x 10 [Gb/s] Sender/Receiver Pairs

Finally, an experiment with all nine 10 [Gb/s] sender/receiver streams (as shown in Figure 1) was executed. The purpose of this experiment was to successfully transmit as much data as possible in parallel through the link given its capacity, and study how well SPEAD behaves under these circumstances.

Results for this experiment are shown in Figure 6. Unlike the previous two experiments, we performed five different measurements for each quantity instead of a single one to present stable results that would smear out other temporary sources of noise, and Figure 6 shows the average quantities only. Like in the previous experiment, UDP aggregate speeds do not scale well after reaching ~50 [Gb/s], and reach a maximum of just over 63 [Gb/s] when individual streams are setup to send data at 9.5 [Gb/s] each (more on this below). Data starts getting lost at around 6.5 [Gb/s] (~56 [Gb/s] aggregated), and the situation quickly deteriorates from 6.75 [Gb/s] onwards, with data loss rates higher than 10%. It can also be seen that the aggregated UDP speed after this point does not scale as quickly as the error rate does, resulting on effective receiving speeds (i.e., the transmission speed of full heaps as measured by the receivers) that continue to decrease. The observed TCP speed, like in the other experiments, naturally agrees with that of a lossless or almost lossless UDP stream.

From our previous experiment of Multiple connections, single interface tests we already expected a sending limit of ~32 [Gb/s] per interface on *vcs17*. The maximum sending rate is however less than double that amount, which was not expected. On the other hand, and not visible on the plot, individual sending speeds during these experiments always had *medconv*'s single interface performing up to expectations. In other words, we were expecting to see a value of ~73 [Gb/s] as the maximum sending speed. A possible explanation for this result is yet more contention happening in vcs17 as an effect of running even more senders on the system. Future experiments (outside the scope of this work) could include trying out different NUMA and CPU bindings and building SPEAD against more static components to avoid cross-node memory transfers when filling L1 instruction caches.

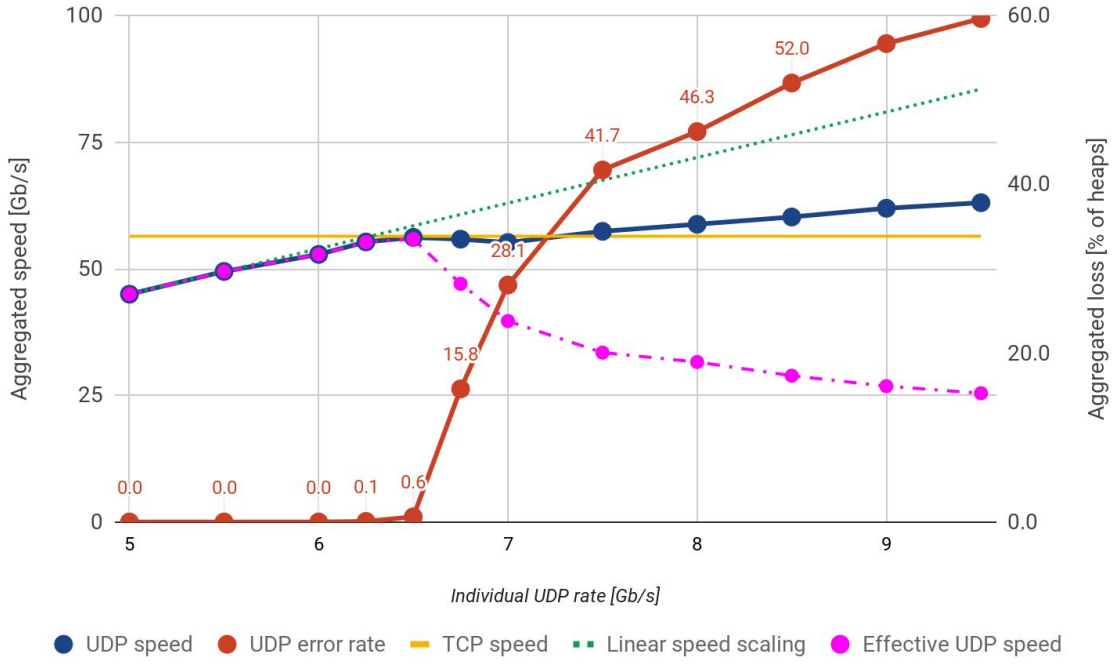

Figure 6: Performance of all nine 10 [Gb/s] connections being sent through the same 100 [Gb/s] link between MRO and Perth. Dark blue points are aggregated UDP speeds, red dots are UDP error rirates, the green dotted green line is the linear scaling of the individual UDP rates, and the

yellow line is the aggregated TCP speed. Like in the previous experiment, UDP aggregated speeds stop scaling and saturate at around 64 [Gb/s]. Aggregated UDP speeds with <1% data loss rates again agree with the aggregated TCP speed with zero loss. The purple line represents the 'effective' UDP transfer speed as observed by the receiver, which is defined as the total volume of heaps received divided by the total transfer time (or, equivalently, the aggregated UDP speed times the UDP error rate).

Conclusions

SPEAD's design is heavily influenced by UDP; however it is up to the application to shape heaps and deal with packet loss. To tackle these issues and also to be able to directly compare achievable UDP and TCP rates, we have introduced a new TCP transport implementation, which offers all the benefits of TCP: transmission reliability, error detections, congestion control and more. These are desirable properties to many applications, and therefore we hope this implementation proves itself useful to the wider community. Changes are currently available under a fork of the original repository [RD04] and will soon be submitted for merging upstream. The initial implementation includes new TCP streaming classes, which are currently not exposed through the python bindings. An improved version of the TCP stream is also suggested that would not require splitting a SPEAD heap into packets on the sender side, and re-assembling them on the receiver side, saving CPU load and memory usage.

We have also presented a study of how well the UDP and TCP transports performs in different situations, including an SKA1-LOW-sized 100 GbE link. Thanks to its clean design and very good and modular implementation, the SPEAD software runs very well in all situations. Ticking at more than 40 [Gb/s] single-threaded, the UDP transport sending stack is limited by parts of the system other than the CPU, like communication buses and network components. The performance of our TCP implementation (ticking at almost 30 [Gb/s] single-threaded) matches that of a lossless or near-lossless UDP transport in all situations, and therefore could be considered for wider usage. This also shows that the implementations of the TCP stacks in the various hardware and software layers involved in these tests provide a near optimal bandwidth utilisation with zero loss.

Even if the SKA would accept a certain loss rate to gain a few percent of performance, the initial tuning and subsequent monitoring of the UDP loss rate would very likely require some kind of auto-tuning mechanism in SPEAD to cope with varying conditions, which are likely to occur regularly on a ~890 [km] network link. In addition our tests showed that the UDP error rate increases very rapidly above a certain limit, which is compatible with the achievable TCP rate. Increasing the UDP sending rate just a little bit above that limit increases the error rate extremely quickly, and therefore we have not been able to observe much benefit when using UDP in terms of performance v/s error. It is also unclear to us how the CSP side of the UDP sender implementation would look like in practice. The software implementation of the sender has a number of adjustable parameters, most notably the throttling rate, which would need to be exposed on the CSP side (in FPGA) in order to be able to maximise the transfer speed, while minimising the error rate. **We thus conclude that the usage of UDP even on**

local networks bears no benefit, but would in contrary require extremely careful optimisation and monitoring in an operational environment, which is both labour intense and risky. In view of these results we recommend to re-evaluate the decision of using SPEAD over UDP.