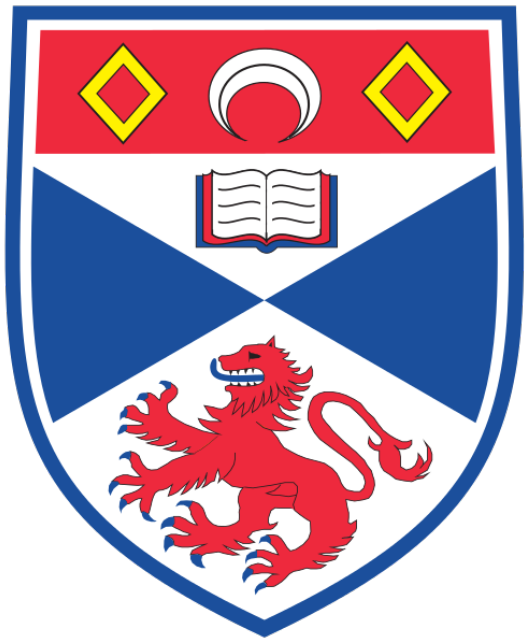
Rust Implementation of the ANSI E1.31-2018 sACN Protocol



University of St Andrews
April 27th, 2020

Paul Lancaster

160007345

1 Abstract

The project aims to create a library for the ANSI E1.31-2018 sACN protocol [3] that is available in native rust.

2 Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. "The main text of this project report is 21,274 words long, including project specification and plan. "In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bonafide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Contents

3 Introduction

Currently within rust there does not exist a library which fully supports all aspects of the ANSI E1.31-2018 streaming ACN protocol [3] (sACN). sACN is a commonly used protocol for transmitting control data for lighting devices (such as those used during concerts) and so the lack of a public-ally available open-source rust library hinders development of lighting devices or controllers in the language. Previous to this project the progress towards creating such a library was an implementation [**?**] which supported the sending and parsing of data using sACN but with many newer features such as Universe Synchronisation and Discovery not available as well as no mechanism to allow receiving sACN data. This project therefore utilised parts of this existing implementation to create a new sACN library which supports ANSI E1.31-2018 sACN universe synchronisation and discovery features as-well as sending and receiving data. This project then goes beyond implementation to thoroughly test the created library to show that it is compliant with the protocol specification and interoperable with a number of commonly used programs which already exist within the sACN protocol space.

4 Context Survey

4.1 DMX, SACN and ACN

4.1.1 DMX512

DMX512 is an protocol used in the entertainment industry for the control of lighting, effects and other devices. It works by daisy chaining devices together into distinct physical chains (called universes) and is a one way protocol. This means that the devices in the line cannot communicate their presence back to the controller so the controller must know about the devices ahead of time and their addresses so it can broadcast packets down the line which the devices then receive and use. The DMX packets are a fixed size and contain five hundred and twelve 8-byte channel (+ a start code) which allows them to control up to 512 different devices on a singular line. A device may support the use of multiple channels to control different functionalities so for example a light with RGB colour mixing may use 3 channels to allow control of the Red, Green and Blue individually. Since there are only 512 channels available on a single universe this quickly imposes a limitation to the number of devices that can be connected together, especially as modern lighting fixtures commonly use upwards of 30 channels each for a moving light with usage of many more not uncommon. The solution to this was previously to simply have more physical lines (universes) and in this way allow more devices to be controlled simultaneously. This

comes with a number of problems however as each new physical line means a new cable coming directly from the control desk. A diagram laying out a typical dmx setup is shown in figure 1.

DMX512 Problems

1. As the control desk is often far from the devices themselves (at the back of the venue whereas the lights/devices are above the stage) it means that many cables need to be run which can be expensive and time consuming.

2. The length of the cable runs can cause signal interference / degradation and DMX does not have any error correction (bad frames if detected are thrown out). There is also no mechanism to allow resending as DMX is a one-way protocol.

3. The protocol only allowing 512 channels per physical line means that a device cannot have more channels than this. This is particularly a problem recently with the advent of complex fixtures which may have many LED's with individual colour control.

4. Each DMX line of daisy chains can only have up to 32 fixtures [4].

5. Each DMX line needs a separate port on the lighting controller which may be limited by the physical space available on the device or cost.

6. DMX requires specialist equipment to handle it such as splitters / merges (merges allow 2 sources to be combined into a single universe/line such as for usage as a backup).

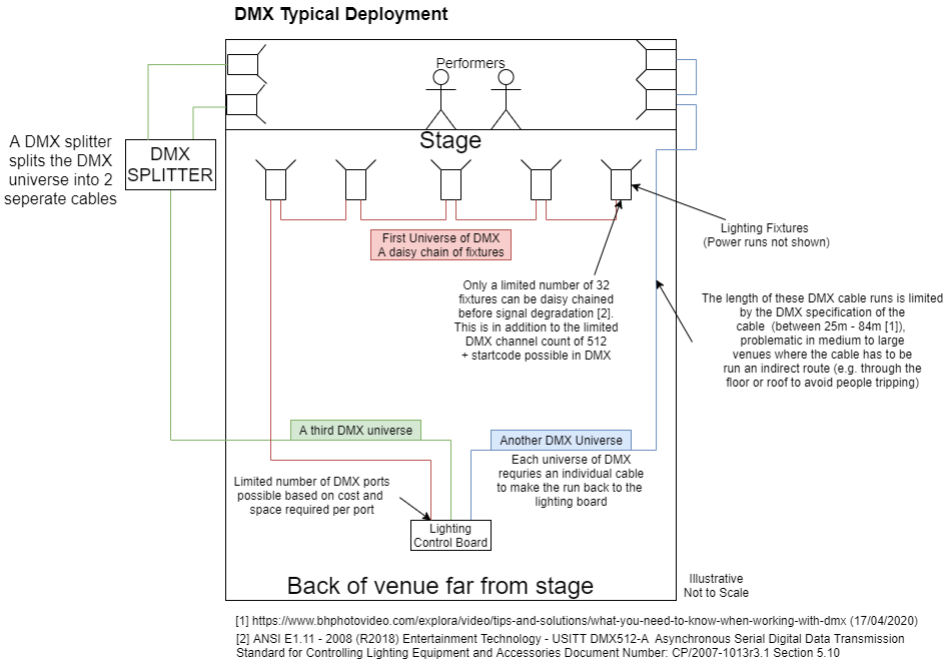


Figure 1: A diagram showing a typical DMX setup which doesn't utilise any IP networking

4.1.2 sACN

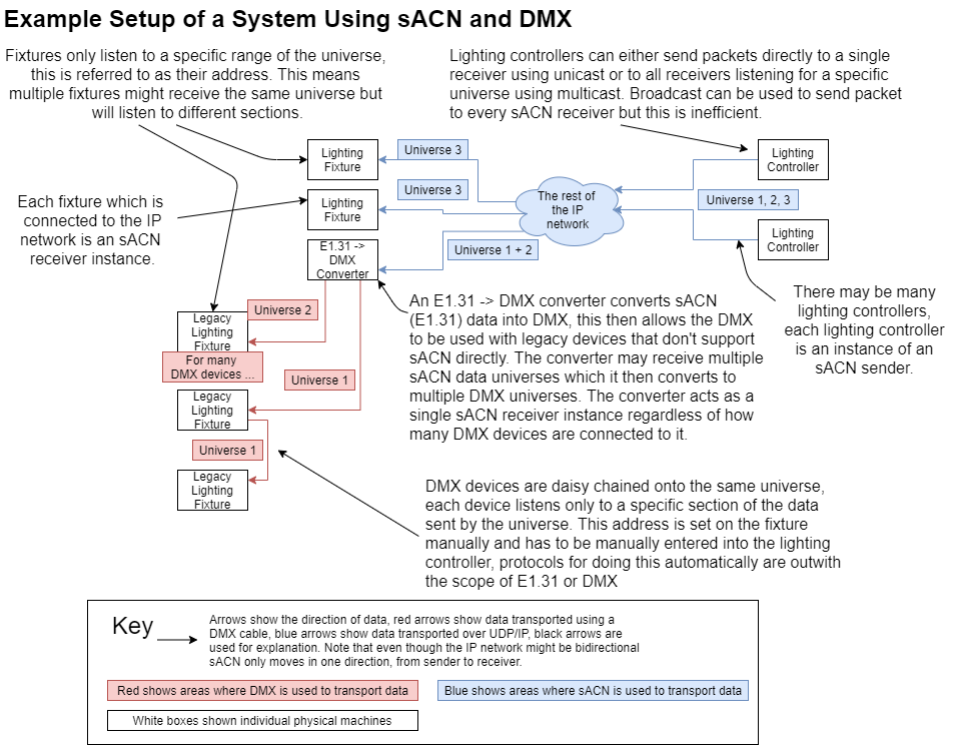


Figure 2: A diagram showing a usage scenario for a system which utilises sACN

One solution to solve some of the problems with DMX is to send it using UDP over a standard IP based network and one of the protocols created to do this is sACN. This allows many DMX packets (and so many universes) to be simultaneously sent using a single network cable from the console and then to be received by the devices. Often for backwards compatibility reasons the sACN is converted back into DMX packets before being sent to the device as most devices older than a few years do not support direct sACN communication but this is rapidly increasing - particularly with higher end professional fixtures. Figure 2 shows an example in which sACN is used for part of the setup with an "E1.31 -¿ DMX converter" used to convert it back to DMX for use with some legacy lighting fixtures.

Figure 2 shows that within a network there are expected to be many sources of sACN in the form of lighting controllers and many receivers in the form of lighting fixtures and other devices. Within this many to many network it is also expected that many different universes will be utilised with some devices utilising the same universes and others utilising a different potentially overlapping set of universes. Within these universes individual

devices all have their own 'addresses' which refer to which section of the universe they are listening to as-well as 'modes' which refer to how many 'channels' (Bytes) are used and what each channel controls on the fixture. This within universe addressing is carried over from DMX and allows interoperability with the older protocol. Actually setting these addresses is done the same regardless of sACN or DMX is used and this is usually done manually (although there do exist mechanisms to do this automatically which are discussed later such as Remote Device Management - RDM). Note that these addresses are independent of the IP addressing. Each device connected to the IP network still has a unique IP address however they might have the same DMX address. In this case both devices would behave identically on receiving an sACN data packet however depending on the IP network setup they might not receive both packets. For example the controller might chose to utilise unicast on the network. This would mean that only the specific receiver being sent to would receive the sACN packets, this can be utilised to allow part of the network to be kept dormant but connected and only used in a backup scenario.

The points below show how a protocol like sACN addresses many of the problems listed above with DMX512.

1. A single network cable (E.g. CAT6) can be used to transport many universes minimising how cables have to be run to the back of the venue.
2. Network cables especially fibre can be run for a much longer distance than a DMX cable without signal degradation.
3. As described below universe synchronisation allows sACN to control multiple DMX universes simultaneously allowing fixtures to use more than a universe of channels.
4. As sACN uses a standard IP networking base as many devices as desired can be connected as long as there are sufficient switches.
5. Only a single networking cable is required from the lighting board into the network meaning the control device can be much smaller.
6. sACN works over standard UDP/IP and so can work onto of commodity hardware switches, routers, network interfaces etc. therefore setting up a more complex sACN network is much cheaper than a comparable DMX setup.

4.1.3 sACN - Universe Synchronisation

A potential problem with multiplexing multiple universes down a single network line is that two universes of data cannot be sent simultaneously, this

is often not a problem for simple devices but for receiving devices that span multiple universes receiving one packet before the another may put the device into an inconsistent state. A similar problem arises if two different devices on different universes want to be controlled simultaneously. ANSI E1.31-2018 provides a solution to this problem in the form of the universe synchronisation feature. This works by data packets containing a synchronisation universe field which can be set to a specific universe. On receipt of a sACN packet with a non-zero universe synchronisation field a compliant receiver won't act on the packet immediately and instead will hold the data for that universe. This data will then be acted upon on receipt of a universe synchronisation packet with the corresponding universe. As data packets for multiple different universes can specify a single synchronisation universe this allows data for multiple universes to be acted upon simultaneously on receipt of a universe synchronisation packet. A diagram demonstrating this is shown in figure 3 with description below.

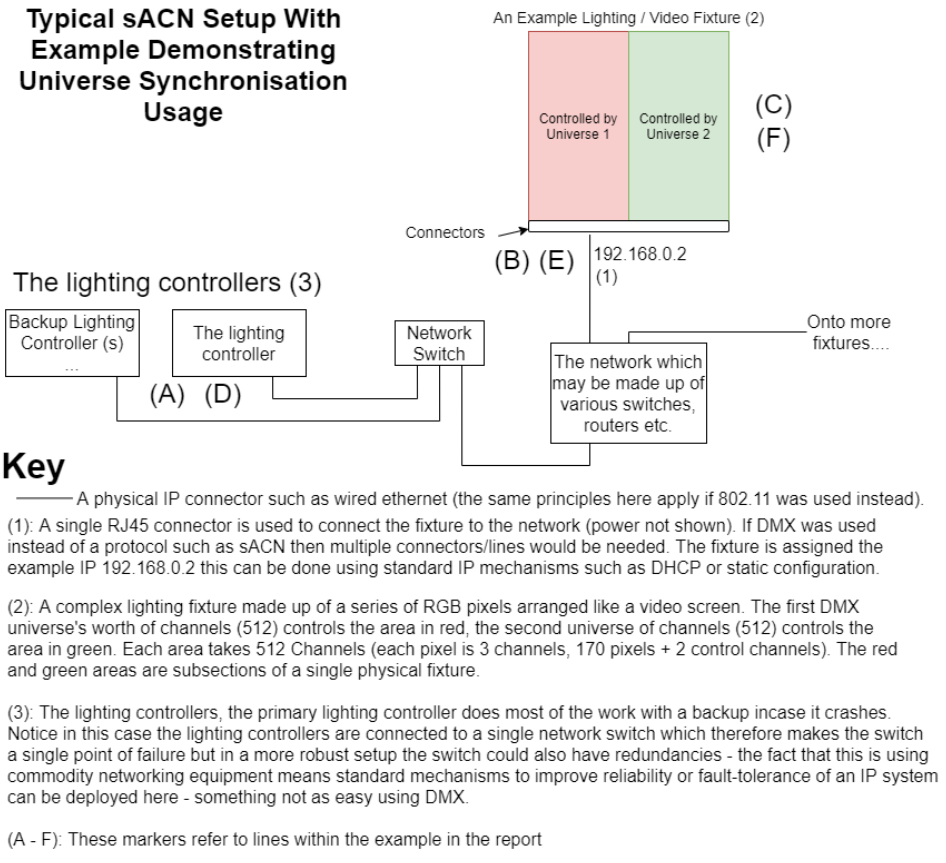


Figure 3: A diagram demonstrating how sACN universe synchronisation allows simultaneous control of multiple DMX universes and its use cases

Usage Example:

The End Goal The lighting controller wants to update the entire lighting fixture (2) to blue simultaneously even though the fixture is split across multiple universes.

Using sACN without any universe synchronisation

(A): The lighting controller sends 2 unsynchronised sACN data packets with 1 going to universe 1 and the other to universe 2. Both tell the fixture to turn to blue. As there is only a single network line these are sent one after the other with a very small delay between them.

(B): The packets arrive at the lighting fixture, due to network conditions such as jitter/delay/re-ordering within the network the spacing between the data packet for universe 1 and 2 has increased very slightly and the ordering has changed. The sACN receiver on the lighting fixture handles the packets in the order it receives them and passes this data immediately up to the actual lighting display and then handles the next packet. Any duplicate packets are removed if detected by inspecting the sequence numbering of the packets.

(C): The command to set the section to blue reaches the left red section before the green section. This causes the left section to turn blue a fraction of a second before the right. This is small but causes a 'screen-tearing' type effect which is subtly noticed by the audience, as this is part of a sequence which is rapidly changing the fixture colour this effect becomes very noticeable as the colour-switch speed speeds up.

Same example using sACN with universe synchronisation

(D): The lighting controller sends the same data packets as in the unsynchronised example but this time with a non-zero synchronisation address of 1 indicating that synchronisation should be performed meaning both data packets should be acted upon together when a synchronisation packet to synchronisation address 1 is sent. The synchronisation packet to synchronisation address 1 is sent after a small (few milliseconds) intentional pause.

(E): The data packets arrive at the lighting fixture and the network conditions of jitter/delay/re-ordering etc. have again changed the spacing within packets and caused the data packets to become re-ordered. The lighting fixture processes both packets but doesn't act on them yet, instead they both wait. Once the synchronisation packet is received the synchronisation address of the synchronisation packet is

checked against the data-packets waiting and as they match the data from both data packets is acted upon and passed up to the actual lighting display together.

(F): The command to set the left and right section to blue reaches the controller together and so both sections are set to blue at the same instant (ignoring nano-second circuit physical delays). This prevents the screen-tearing type effect and means that the entire fixture changes colour at the same time. This example could have also applied to 2 completely separate fixtures which would have both independently waited on the synchronisation packet and then when they received it they would have acted together. Note that in this case 2 universes were used for a fixture but any number of universes could have been synchronised and it would still only require a single synchronisation packet per group of universes to synchronise so therefore O(1) scalability as the number of universes per fixture scales.

4.1.4 sACN - Universe Discovery

sACN allows sending on up-to 63998 universes with each universe having a unique multicast address. Any of the universes can be used by any source and so in initial versions of the protocol such as ANSI E1.31-2009 [14] the only way to learn which universes were in use were either to have prior knowledge or to scan every single possible address and listen for packets. This is very inefficient and impractical in a real-system especially as in the time that a universe was last scanned another source might have joined and started transmitting. Universe discovery solves this problem through the universe discovery mechanism. This mechanism works by having a reserved universe of 64214 (as defined in ANSI E1.31-2018 Appendix A) on which sources send universe discovery advert packets. These packets contain a list of universes that the source sends which is referred to as a universe page. Each page can hold 512 universes and so therefore a source may send multiple discovery packets each with a different page that the receiver can then put together to build up a complete list of universes that the source is sending. To allow a receiver to know when all the pages have been received for a given source each universe discovery page has a numbering which increases sequentially with the number of the last page expected included. By having multiple pages it prevents the protocol being required to send large packets on the network (size limited by page size not by the much larger number of possible universes). This is advantageous as it prevents problems with sending large packets such as causing a-lot of fragmentation at the link layer which will fragment packets into frames that are the size of link-layers maximum transmission unit (e.g. 1500 bytes for Ethernet [24]). A figure showing a typical usage of universe discovery is shown in figure 4.

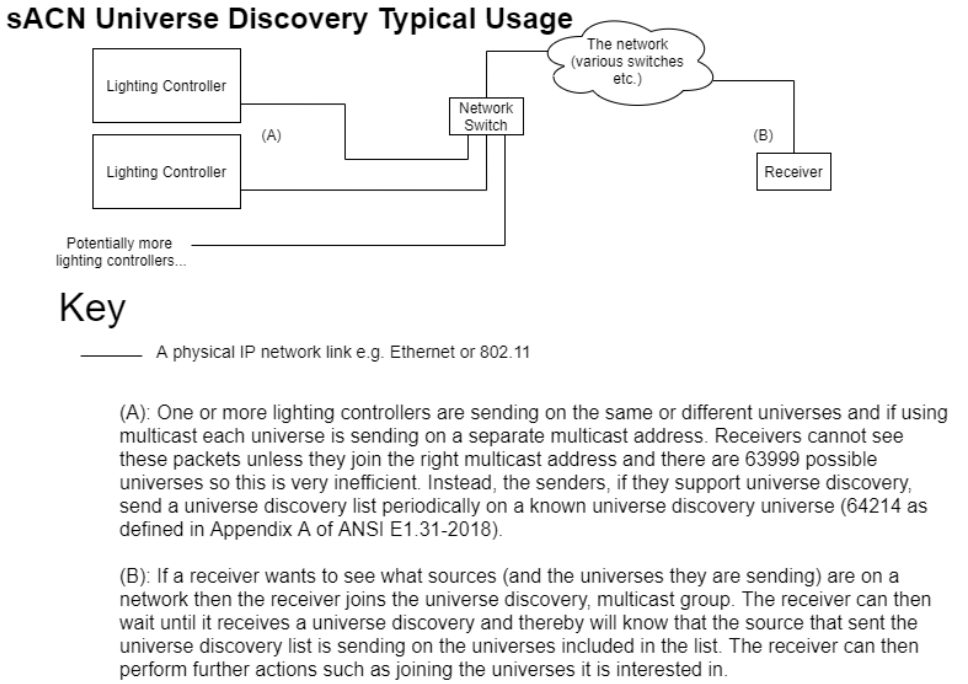


Figure 4: A diagram and explanation demonstrating how universe discovery can be used by a receiver to discover sources

It should be noted that by default a receiver will receive and act on data packets from a source even if it hasn't been 'discovered' yet. This means that the number of sources communicating over multicast is completely transparent to the receiver meaning it places no limit on the number of allowed sources which allows the system to scale if required. Put another way this means that if a receiver is only interested in receiving from universe 1 then it only needs to listen to universe 1 and can completely ignore other universes and universe discovery.

4.1.5 Network Layers / Transport Modes

sACN falls within the application layer of the 5-layer network stack as shown in Figure 5. This is because it sits on top of UDP (layer 4) and IP(layer 3). As UDP is used as the underlying transport protocol it means that there is no guaranteed delivery of packets. The protocol itself also doesn't provide this which means that data send by a source may not reach a receiver and there is no way for the source to know within the protocol scope. This loss of guarantee comes with the advantage that there is less packet overhead and no hand-shake is required meaning data can be sent immediately. The use of UDP also avoids many of the problems associated with session transport

protocols like TCP such as lost packets significantly reducing throughput due to the congestion control mechanism. Theses trade-offs fit the expected usage of the sACN protocol as they minimise latency which is vital in a real-time event/lighting system.

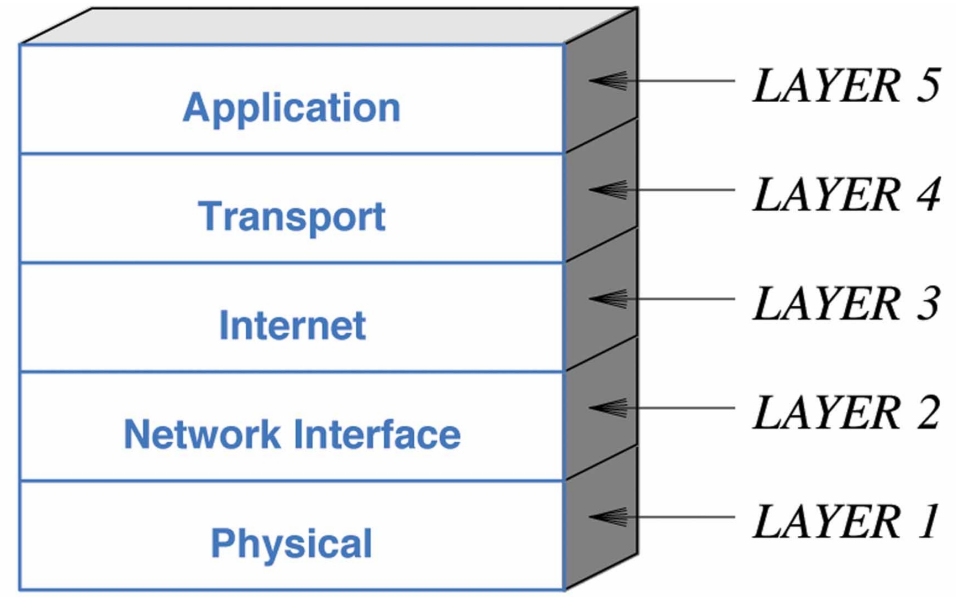


Figure 5: Image showing the 5 network layers, image from [34]

The usage of UDP additionally means that packets can be delivered in any order, this can cause random jumps in data on the protocol which is noted within the specification to be problematic if this is used with a moving head lighting fixture as it effects the predictive algorithms used (ANSI E1.31-2018 Section 6.7.2). To reduce this happening the protocol uses sequence numbers to allow out of order packets to be discarded. It is important to note that because a packet might have been lost the protocol doesn't attempt to wait for packets which haven't been received yet and instead always acts on the most recent data (with regards to sequence number), discarding any old data received. This keeps the latency of the system low and prevents slightly out of order packets causing unexpected jumps back and forth in the data. As the sequence number field is only 1 byte in length it is expected to wrap around frequently, therefore the sequence numbering mechanism accounts for this by looking at the difference between the last and current sequence numbers as oppose to the numbers themselves directly. This difference is then checked if it is within the range of (-20, 0] (greater than -20 exclusive, less than or equal to 0 inclusive). If it is within this range then the packet is rejected otherwise it is accepted.

This allows the sequence number to wrap around without packets being incorrectly discarded. It also means that if a packet with an unexpected sequence number is received it allows the system to quickly (within 20 sequentially numbered packets) start accepting packets again which minimises latency. Sequence numbers are evaluated/incremented separately for each packet type and within packet types separately for each data-universe in data packets and synchronisation-address in synchronisation packets. This means if a packet with an incorrect sequence number is received it will only effect that type of packet and universe allowing other universes/packet-types to continue as normal.

In its current 2018 state the protocol specifies operation over IPv4 and IPv6 using 3 different IP communication modes. The first mode is unicast, this is where a source sends data directly to the receivers IP and this means that any data sent by a source must individually be sent to all receivers for them to see it. The next mode is broadcast, this is where the destination IP is set to a special broadcast IP which causes all receivers to see the data. This mode means that a sender doesn't have to individually send to each receiver but does mean that there is the potential to flood the network with these broadcast packets with all receivers getting packets even if they didn't want them.

The final IP communication mode utilised by the protocol is IP Multicast. This is the default mode used and works by receivers joining ip multicast groups which senders can send to with only the receivers that joined the relevant multicast group seeing the sent packets. This minimises the packets transmitted to uninterested receivers with packets only routed to receivers that have joined the relevant multicast group but without the senders having to know the address of each receiver. Within the sACN protocol each universe utilises a different multicast address and therefore all receives and senders can use a specific address to receive from / send to a specific universe. As this mechanism does not require the receiver(s) or sender(s) to know about each-other ahead of time it improves the scalability of the system as a sender requires the same amount of processing power to send a single universe of data to one receiver or to a thousand.

Multicast Address Assignment To allow usage of multicast within an sACN network there are standard multicast addresses defined for each universe so that a receiver and sender know where to receive/send data. For a sACN universe IPv4 multicast address the first 2 bytes are always 239 and 255 respectively. The 3rd byte is the upper, most significant byte of the universe (when the universe is expressed as a 2 byte unsigned number) and

the 4th byte is the least significant byte. The IPv6 mapping is similar with the least significant 16 bits used for the universe number. This is shown in figure 6 below.

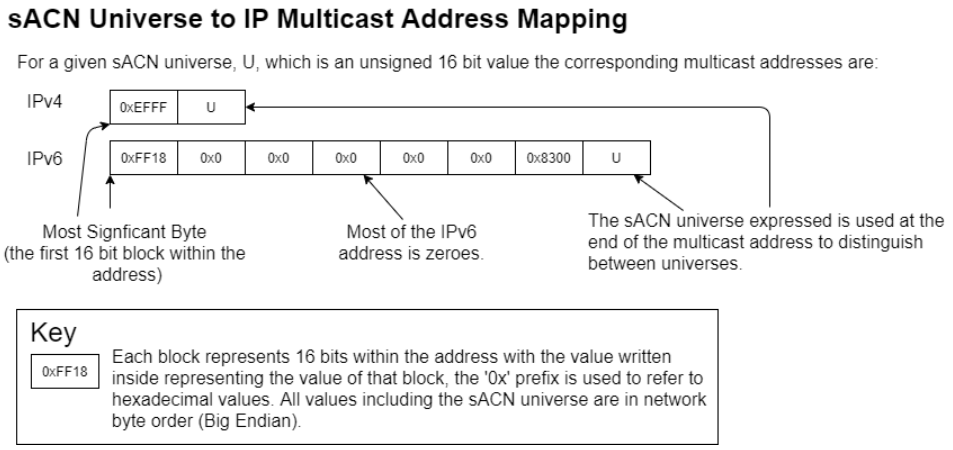


Figure 6: The mapping used from an sACN universe to an IPv4 or IPv6 multicast address within an sACN network

As specified in the internet engineering task force (IETF) RFC 5771 [35] all of the IPv4 multicast address fall within the 'Administratively Scoped Block' as specified in IETF RFC 2365 [36]. This then clarifies that the sACN multicast address fall within the IPv4 Local Scope range (6.1). These multicast addresses are reserved for usage dependent on the specific local network within which they are deployed. This limits the sACN usage to a local dedicated network and not for use on a wide area network (WAN) for as the public internet. As discussed previously this fits with the other design decisions made which based the protocol around usage on a private, isolated dedicated network.

The IPv6 multicast address assignment starts with 0xFF in the first byte, this indicates that the address is a multicast address. This is then followed by 0x18 which can be broken down as per section 2.7 of [37] into a flag value of 1 and a scop value of 8. This flag value mean that this is a transient address which indicates that it isn't statically assigned by the IETF and may change in future. The scope value of 8 indicates organisation-local scope which has a similar reasoning as the local-scope used for IPv4 meaning that this address is only valid within a specific environment / group of networks. Therefore the sACN protocol is not aimed at usage on a WAN regardless of IPv4 or IPv6.

4.2 sACN Packet Structure

As sACN is a subset of the wider ACN family of protocols it utilises a common standard ACN header. By using the general ACN header it allows other ACN protocols to be used on the network alongside sACN without conflicts, for example it is later discussed in the relate works that ANSI E1.33-2019 (remote device management) and ANSI E1.31-2018 can be used together to increase the available functionality in the system. The packets are split into layers with each layer handling a different part of the packet. As there may be multiple possible layers the packet contains multiple 'vector' fields. These fields contain predefined values which tell the receiver about what the data will be. The structure of the packets is described directly within the protocol as a table and has also been expressed as a diagram in "Packet-Structure.pdf".

4.3 Critical Analysis of the sACN protocol

ANSI E1.31-2018 sACN over a purely DMX network provides a solution to a number of problems as discussed above but also has its own problems.

In the universe synchronisation example above the mechanism allowed the update to happen simultaneously on a single fixture however this came at the cost of requiring 1 more network packet than without synchronisation (2 data packets + 1 synchronisation packet vs just 2 data packets). This is potentially significant on a network with hundreds of simultaneously controlled fixtures and could lead to congestion. The reliance on another synchronisation packet has a further problem which is that it is also subjected to the network conditions. This can lead to the synchronisation packet being lost, in this case the fixture won't act at all (which in some situations might be better than it half acting). Another problem comes in if the synchronisation packet is reordered, if it arrives before the data packets then they won't be acted on at all (unless another sync packet is sent). If the sync packet arrives between the data packets then only the first synchronised data packet will be acted on, this means that even with this mechanism fixtures can end up in an inconsistent state with only one universe acted upon. As noted within point (F) of the synchronisation example synchronisation can also be used between fixtures however this suffers from the problem that there will still be a delay caused by the difference in timings taken for the synchronisation packet to reach each fixture.

Another potential issue with the protocol is that it provides no protection from malicious or malfunctioning sources taking control of the system. This makes isolation, preferably physical, of the network vital and so ANSI E1.31 sACN is commonly used on networks dedicated to lighting protocols.

This also helps reduce the issue of variable transmission latencies as these networks are likely to be fairly simple. Even with isolation from malicious users the sACN protocol is still vulnerable to problems related to byzantine failures where devices fail but rather than doing so cleanly instead produce random values which are interpreted by devices on the network as intentional and can cause the system to act unpredictably. These failures are not-uncommon in networks using cheap, knock-off devices which might not be fully compliant with the protocol even if they work most of the time.

The protocol also suffers from the same problems that many protocols do related to trying to maintain backwards compatibility, particularly with DMX. This imposes a number of limitations and inefficiencies. One example of this that each sACN packet sends a single universe limited to 512 channels, this is far less payload than the packet could actually hold, even if 2 universes were sent in one packet it would half the number of packets required and produce packets of size 1150 bytes (current size: 637 bytes + a universe (513 bytes)) which is less than the MTU of many common link-layers e.g. Ethernet. In addition to this the concept of universes themselves limits the protocol as problems due to devices being unable to lie across universe boundaries have been carried over into the protocol and solved. For example universe synchronisation is at its core a solution to only being able to send a single universe per packet however if you could send more than a single universe then it wouldn't be required at all. An example of how this could work for example is that a packet could be sent for every individual device or group of devices with variable parameter counts meaning redundant data isn't sent. This would still be subject to fragmentation from lower layers e.g. to fit within the MTU of the link in use.

The protocol layers (UDP + sACN) also add a significant amount of over-head, for a full universe of data which takes up 513 bytes (512 DMX channels + a start-code) the packet size is 637 bytes meaning an overhead of 124 bytes, corresponding to 19.5% and if the universe is only partially full the ratio of overhead to actual data gets worse (1 byte of data + 1 start-code leads to a packet that is 98% overhead).

4.4 Related Work

The ANSI E1.31 sACN protocol was originally specified in the document ANSI E1.31-2009 [14]. This represented the base version of the protocol without any universe synchronisation, universe discovery or discussion of operation with IPv6. Since then it has been revised in 2016 (universe synchronisation and discovery) [15] and again to its current latest version in 2018 (IPv6). The future of ANSI E1.31 is still being actively developed and

discussed [16] with the direction of the ACN eco-system being focused on supporting communication from receivers back to sources. This would allow sources to detect receivers and optionally configure them remotely. This has significant use in cases where there are a significant number of receivers or the receivers are located in hard to reach areas (such as high up in an arena lighting rig). Within traditional DMX systems this is supported using the remote device management protocol (RDM) as described in ANSI E1.20-2010 [19]. This protocol allows a number of configuration options such as remotely setting the DMX addresses of fixtures and has proven usage within real-world environments. A recent iteration of this protocol is an IP version known as RDMnet [17] which is ACN based and allows discovery and control of receivers over a network. RDMnet as a fairly new protocol and so is still in the process of being taken up by vendors but has strong support from ETC (a large lighting company [20]) in the form of a maintained open source implementation of RDMnet in C++ [18]. RDMnet while a good forward step still suffers from some of the problems related to sACN in that it is still based on DMX. Similar to what happened within the traditional networking world it is very likely that eventually the entertainment control ecosystem will move to a completely IP based system. This would have significant benefits as fixtures could be configured completely remotely (using standard mechanisms such as SSH / Telnet / Webservices) and rather than requiring specialist DMX control at all it could all be commodity network hardware which is cheaper and more flexible in its usage. This would also allow many of the advancements in other areas of computing to brought into the industry such as advances in IoT

The ACN based family of lighting control protocols aren't the only protocols that allow sending DMX data over an IP network. Another widely adopted protocol is ArtNet which at time of writing is in its 4th version. Unlike sACN on its own ArtNet allows discovery of receivers, remote configuration and transporting remote device management data (receiver meta-data) [**?**] in addition to sending data. ArtNet therefore has taken the strategy of being a larger protocol which covers multiple use-cases within a single protocol as opposed to the more split up ACN strategy (ArtNet v4 is roughly equivalent to ANSI E1.31-2018 and ANSI E1.33-2019). While they are developed independently the ArtNet v4 standard does include the ability to interoperate with sACN. In this mode ArtNet is used to configure and control sACN devices and then sACN is used for sending data [12, Pg. 3].

There are a number of existing implementations of sACN in rust however none are fully compliant with the protocol as specified in ANSI E1.31-2018. One of the most complete is [2] which was used as the base for this project. As this is hosted on github it can be seen that while there are a number of forks (6 at time of writing) no public fork has any further progress which

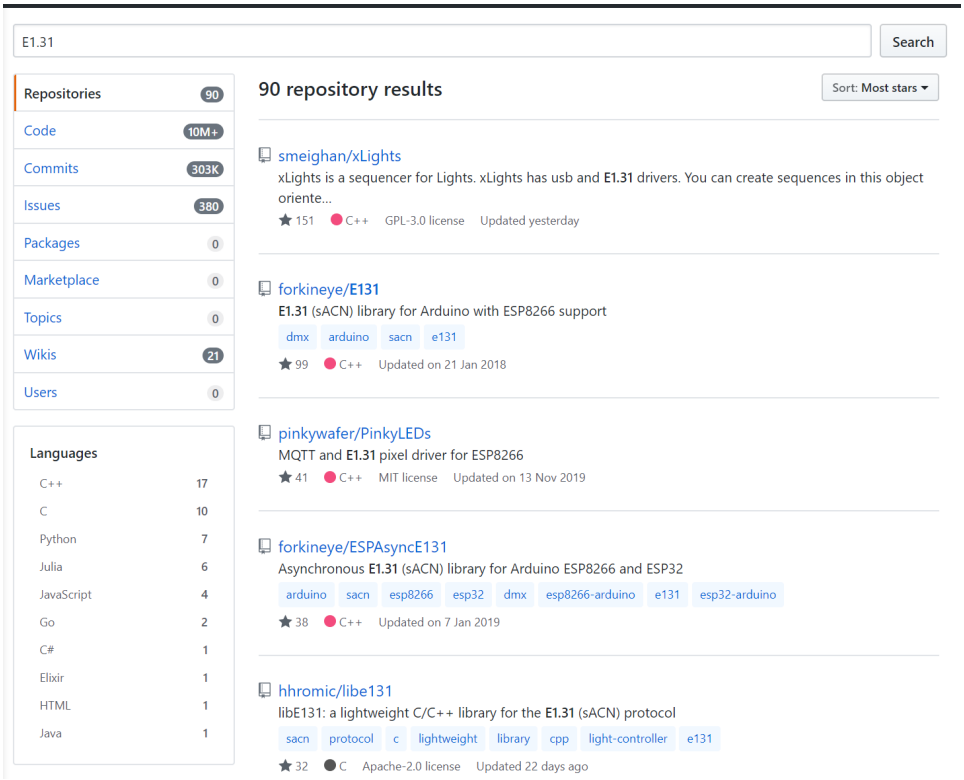


Figure 7: A search of repositories on github with the search term "E1.31" as of Jan 2020

leads to the conclusion that this is the most complete open source rust implementation available. Note that this implementation appears in a number of places such as [13] but this is still the same implementation. This existing implementation provides support for parsing sACN packets and sending sACN data packets using multicast. The existing implementation however does not provide support for universe synchronisation/discovery, sending data using a mechanism other than multicast or receiving data.

Implementations of sACN exist in multiple languages, at the time of writing (Jan 2020) a cursory search for E1.31 repositories on github reveals the most prevalent libraries being in C++ and C as shown by Figure: 7. An example of one of these projects is [6] which allows both sending and receiving of sACN packets but does not support universe synchronisation or discovery.

5 Requirement specification

The project was split into the following list of primary and secondary functional and non-functional requirements
Primary, Functional Requirements

> Allow sending and receiving DMX (or other start-code data) over sACN.
>
> Support the sending and receiving of synchronised DMX data through the universe synchronisation feature.
>
> Support universe discovery with adverts for sources and discovery for receivers.

Secondary, Non Functional Requirements

> Demonstrate a deployment of the library into a real-world system to show its compliance with the protocol by showing interoperability with other compliant devices.
>
> Provide support for Windows 10 and Fedora Linux systems.
>
> Support multiple IP transmission modes - Unicast, Multicast and Broadcast.
>
> Support multiple IP versions - Ipv4 and Ipv6.

The intended user for this library is a software developer developing applications that utilise the sACN protocol. It isn't designed to be used directly by an end user as it is just a library which needs to be used in code to actually perform any actions. This means it needs to be able to be understood and utilised by someone who is familiar with general software engineering and the main ideas of sACN. This makes technical documentation of the project code such as comments, API explanations and examples a vital part of the project as otherwise developers won't want or be-able to use the library.

SACN is commonly used in heterogeneous device environments with a mix of different operating systems such as Windows and Unix. Therefore to provide support for as many devices as possible a few additional non-functional requirements were made; The library therefore should have support for both IPv4 and IPv6 as well as unicast, multicast and broadcast in both windows and unix environments. Backwards compatibility with the existing library was abandoned due to the incomplete nature of the library and to re-use it would require significantly forcing the implementation of the new library into confusing patterns to allow usage of the new Synchronisation and Discovery features.

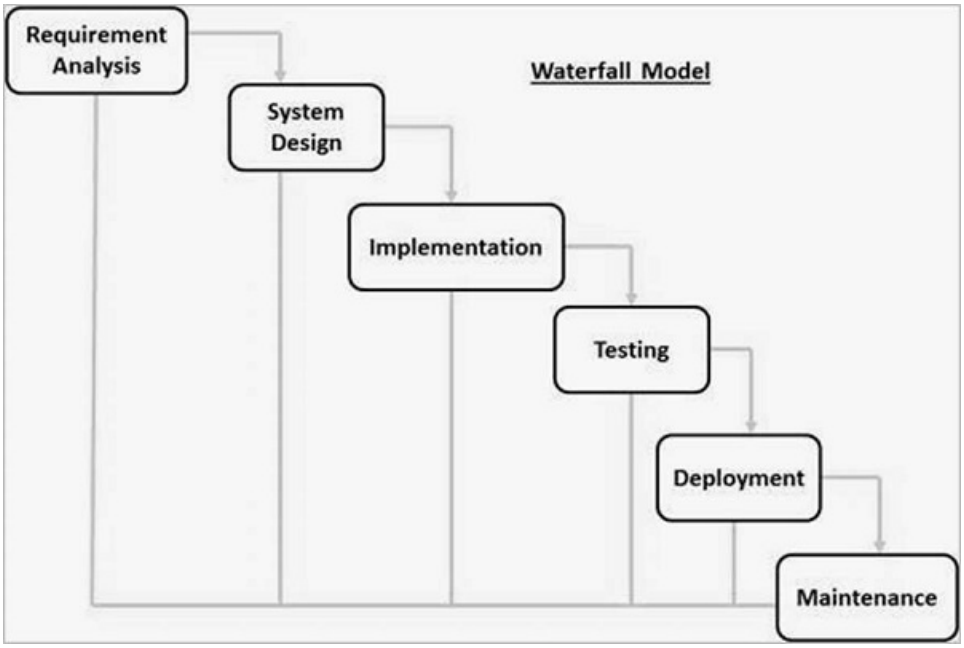


Figure 8: A diagram showing the waterfall development process, [[23]]

6 Software Engineering Process

A waterfall based process model was used for the development of the program. In the waterfall method there are several distinct phases of the project as shown in figure: 8 which follow on from each other with loops back possible if a problem is found at a later stage. This development approach was chosen as it has a very clear structure which allows easy to manage distinct milestones so progress through the project can be more easily tracked. The main disadvantage of this process is the inflexibility, if something major needed to change it would be difficult to adapt the project. This is not a problem for this project however as the project is based on a clearly defined specification provided by the protocol specification and the domains were clearly defined at the start. This means that this inflexibility isn't an issue and so therefore choosing the waterfall method is a suitable software engineering process for this project.

The waterfall model can be clearly seen throughout the development of the program. The first phase of 'requirement analysis' is the protocol specification itself as it clearly lays out the goals of the protocol and what it is required to do. On top of this there is the project goals which were defined around the protocol specifically for how much of the protocol this specification should implement for example universe-synchronisation, IPv4/IPv6 support, Unix/Windows support etc. When taken together this gives a clear list of requirements as so allows moving onto the 'system design' phase.

The system design phase is where the requirements are turned into a technical plan for how they will be implemented. Part of this comes from the protocol specification itself as it describes how each bit of a compliant implementation should behave and so therefore the design can be based of this. In addition to this part of the design is based on the existing base implementation. The design is detailed in more detail later but in general is based around distinct receiver and sender implementations with all the communication being 1-way from sender to receiver. This allows both sides to be developed in relative isolation with the protocol providing the only communication between them. This isolation makes testing easier as there are 3 distinct areas to test: the sending mechanism, the receiving mechanism and the protocol packet structure.

6.1 Implementation, Testing and Deployment Phases

Once the software design is established the next steps are the Implementation, Testing and Deployment phases. Within this project this represents the largest part of the work. The implementation phase is one of the biggest in this project and represents the actual creation of the code as discussed in more detail in the Implementation section. As part of the engineering process there was an amount of looping between the implementation and testing phases. This was done as each part of the code was implemented (for example adding universe synchronisation) which was then tested by creating some initial tests to check that the design for that section has been implemented correctly. Then the implementation phase was revisited to either fix a discovered bug or to implement the next section. This looping is similar to the way that a test-driven-development methodology might work however the waterfall methodology described here is distinct as the implementation is written before the tests. This is distinct from an agile process as the design of what will be created in the end stays the same throughout.

The implementation is known to be complete when all the functionality specified in the design has been implemented. In this project this is represented by data sending, universe sync and universe discovery all being implemented on both the sender and receiver. At this point the project moves into the testing phase. The focus now becomes on verifying that the implementation is correct with respect to the design (compliant) using a holistic view with all parts put together as-well as ensuring the documentation matches the actual behaviour. During this stage it is possible that bugs or areas where the implementation isn't compliant with the protocol specification may be discovered. In this case the focus will move briefly back to the implementation stage to fix the problems before progressing

back through to the testing phase. It is possible at this phase that a design problem is encountered, for example if it was found that the structure of the program didn't support a functional or non-functional requirement. If this happens then at that point the engineering focus would move back to the design stage and as per the waterfall model the focus would then continue through the process of the implementation and testing phases. The testing phase is signalled as complete when there is sufficient tests that verify that all functional and non-functional requirements have been met. What counts as sufficient is discussed in more detail in the testing section.

The next phase is the deployment phase, within a larger/real-world development project this is where the finished and tested code is given to users to use. As an analogue for this in this project this is shown by the real-world acceptance tests. These tests fall across the boundary of the testing and deployment phases as they both verify the system works but also show that it is sufficiently mature that it could be deployed into a real-system and utilised. For this project the intended end user is a software developer creating a program which allows usage of the E131 protocol. Having an actual developer use the library is beyond the scope of the project. As an alternative the demo sender and receivers act as an example of a possible deployment. These demo programs are then demoed by interacting with a real-system and this is shown to someone who actually works in the field (see acceptance testing). Passing these tests indicates that the project has reached the stage of actually being being deployed. As part of this stage it also includes the packaging of the project so that it can be used by developers including the finalisation of documentation and a list of dependencies, once this is complete and the demo programs have been packaged the deployment stage is complete. This is the point at which the scope of the project ends as the final 'deployment' is marked by the final submission.

The final stage is the maintenance stage, this falls out with the scope of this limited time-period project however in a real-world project this represents the process of users reporting bugs, problems, feedback and developers looping back to one of the various stages such as design, implementation or testing to verify the problem and implement a fix. While not part of the project directly it is hoped that the library will be able to be contributed back to the community e.g. through the rust cargo repository and GitHub and by doing so the maintenance stage can begin with me and community acting as the maintainers.

6.2 Reflection on Methodology Used

The approach fit the project well as it made it clear which stages the project was focused on (implementation, testing, deployment) with the previous stages (analysis, design) clearly shown by the protocol specification. The methodology did require increased up-front work as implementation could not begin until the analysis and design states were complete. This up-front work came in the form of the initial documentation for the project such as the DOER list of objectives and as this is required anyway this isn't a problem for this project. The methodology also meant that there was the risk that too long could be spent on one stage which delays further stages and therefore the entire project doesn't reach the deployment stage by the fixed deadline. This meant that a time-line had to be created early on to mark when various parts of the project would be complete so that progress could be tracked. This was attached as part of the project in the 'Objectives with times.txt' file and its creation and modifications are shown by the git-version control which shows how it changed. This was later superseded by minute notes at weekly meetings where the project and its progress were discussed. Taken together these show how the project has developed and how the requirements have been changed from those originally proposed due to time-constraints.

A high level view of the development of the project over time is shown in Figure 9. This shows that the waterfall methodology was followed starting with the deployed existing library which is moved into the requirement analysis stage as new requirements are set as discussed in the requirements section. The requirements are then turned into a design using some of the structure provided by the existing base implementation and a set of milestones created. As discussed above the project then enters its main stages of implementation and testing which loop around as sections of the program are created, tested and debugged. It can be seen that on the 12th of January the project had to loop back 2 steps to the design stage and this was due to the non-threaded structure of the program being insufficient to allow the periodic universe discovery adverts and so the design had to be changed to a threaded structure to allow this. This change was a fairly minor change and so the project quickly got back on track. The implementation of this new design and testing was then performed as part of the next 2 steps as per the waterfall model. This was then followed by a stage of further testing indicating the start of the test phase. This phase also included instances of steps back being required such as the bug found on March 7th which required implementing a new OS specific socket handling mechanism to fix. The testing phase then continued up until the code was ready to be demoed in the real-world environment which marks the transition from the testing phase into the deployment phase which continues until final submission.

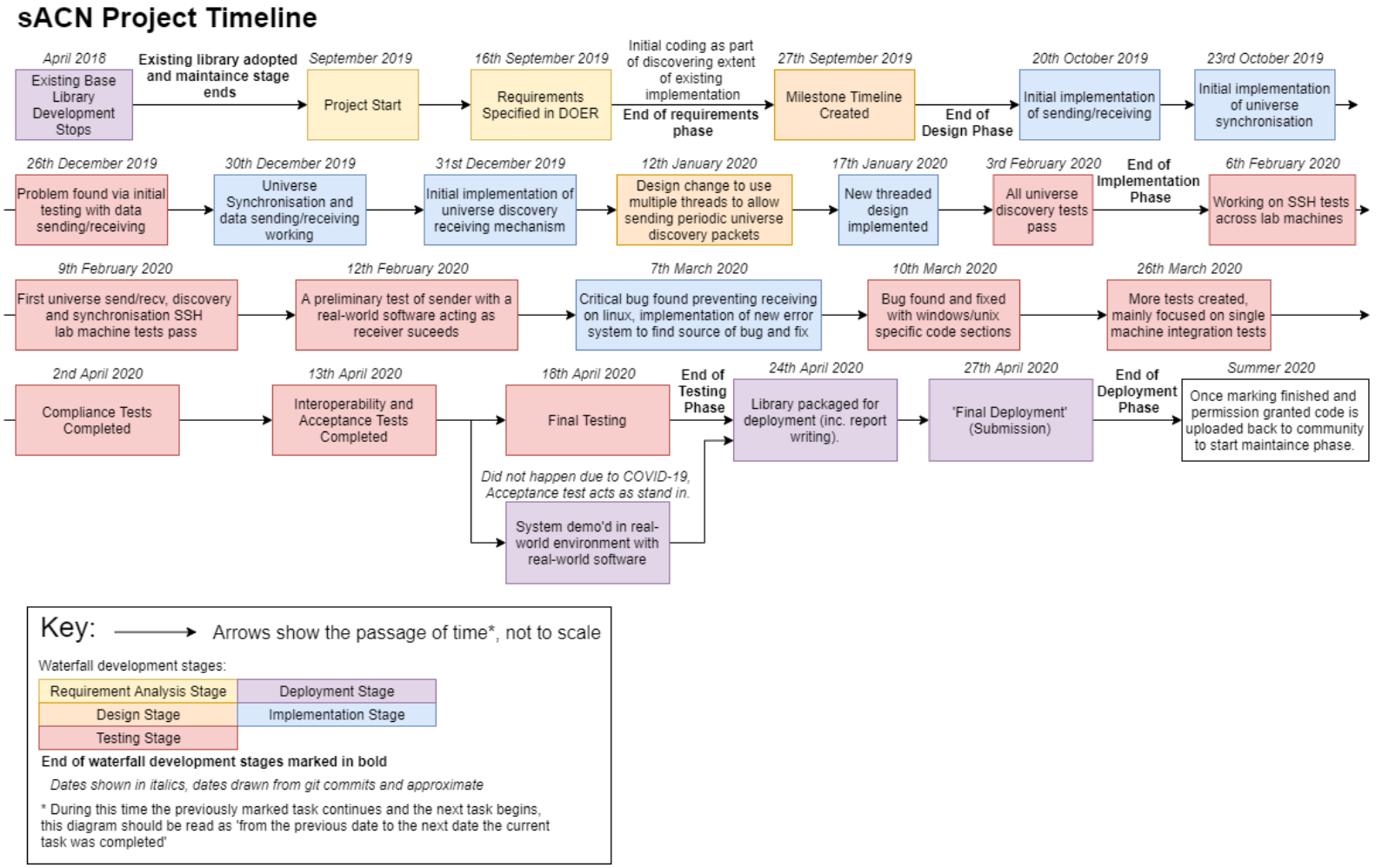


Figure 9: The development of the project over time with the water-fall methodology stages marked. As this may be too small to read in place it is also included as "Development-Timeline.pdf"

7 Tools & Technologies

7.1 Language: Rust

Rust [11] is a compiled memory safe language with no garbage collector. It is extremely fast with near C/C++ like performance [21] but with a much stricter compiler that guarantees memory safety. As Rust has no runtime due to no garbage collector it is applicable to high performance applications making it an ideal language for an ANSI E1.31-2018 sACN device which are often utilised in environments such as concerts where real-time performance with minimal latency is vital to keeping lighting devices in sync with sound. The memory and thread-safety guarantees provided by rust are also ideal for many of the application that sACN are utilised in. These guarantees make applications in rust significantly more robust as they exclude an entire class of bugs and security vulnerabilities. This is useful for devices used in the entertainment industry especially as they are often 'show critical' meaning that if they stop working it could ruin a large event with significant financial implications e.g. if the lighting went out at a big concert and people demanded refunds.

This project was developed and test for rust compiler version 1.40.0 or later.

7.2 Dependency Management: Cargo

In terms of tooling the project utilised a fairly standard rust development tool-chain based around the cargo package manager. Details of how to use these tools relevant to the project are provided in the usage.pdf. The project was developed and tested using cargo version 1.40.0.

Within rust libraries/packages are referred to as crates and the management of dependencies is handled by Cargo. This system allows fetching of dependencies as required during the build stage and includes automatic handling of sub-dependencies etc. In addition to this the Cargo system provides many commands related to testing, compiling, documenting and creating rust applications.

7.2.1 Run

The cargo run command allows checking of the rust code for compile time errors, fetching of dependencies, compiling/linking of the produced rust binary and finally running the produced binary all within one command. This greatly simplified development as there were no Makefiles or similar to manually manage and the code can easily be moved to a new system for development/testing as required libraries are fetched automatically.

7.2.2 Docs

As discussed the targeted end user of the library is a software developer. This makes good comprehensive documentation vital so that developers know what each part of the code does and as this project is expected to eventually be released open-source having good documentation allows new library developers to come in and maintain/expand the code base.

Documentation within the project is done using the rust-doc system which is included as part of the rust/cargo development package. This library is very similar to those found in other languages such as Javadoc for Java which work by having documentation embedded within the code which is then transformed into a HTML web-page to provide the documentation for the crate. As it is directly embedded into the code this makes it less likely that it will fall behind as the documentation and code are close together and a developer can change both simultaneously without having to work across multiple different documents. As this is part of the Cargo system it allows the library to packaged up along with its generated documentation automatically so that when it is distributed to the cargo repository the documentation can be readily accessed alongside.

7.2.3 Test

One of the verification methods used with the library is unit tests, these are small self contained tests which can quickly run to verify that a small part of the program behaves as expected. Rust comes with a built-in form of unit testing through usage of the cargo test command. This automatically finds all tests within the code as designated by the test annotation and runs them producing a list of which tests passed and failed. This also includes other tests such as examples in the documentation which helps to prevent problems where examples are forgotten about and as the development progresses become depreciated or broken. Cargo test was therefore an important part of the project during development and remains an important part once the code is in the maintenance phase.

7.3 Debug Tools: Wireshark

As a network protocol sACN packets can be inspected on the network and the main tool used for this was Wireshark [29]. This was a crucial debugging tool as it allowed checking that packets were formatted and contained the data that was expected. It was also used to trace a number of bugs related to sending/receiving as it can be used to ensure that packets are actually reaching the destination or if not find where they are lost. To allow working with sACN specifically wireshark has built-in support for the protocol aswell as displaying the internal DMX payload, these settings are not enabled

by default so in addition to enabling the ACN protocol the following settings were used shown in figure 10.

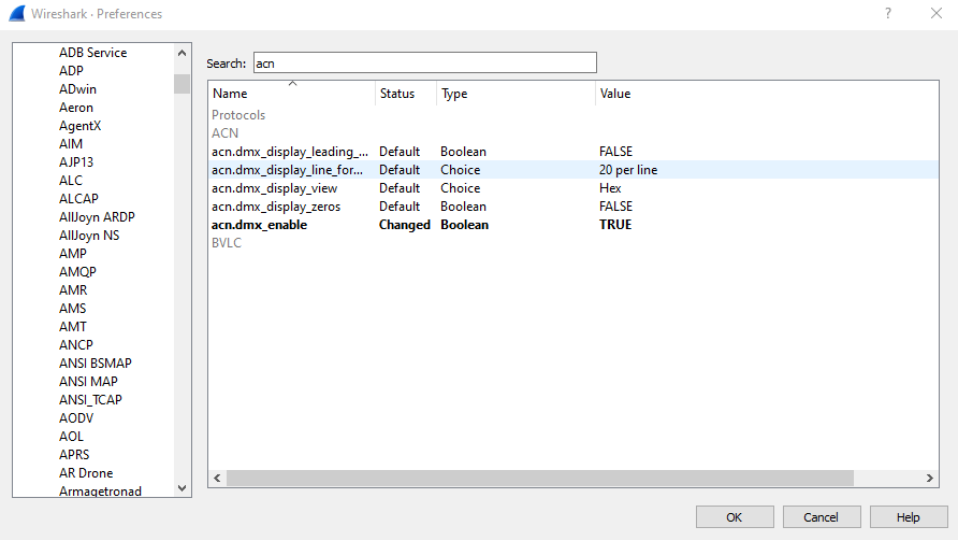


Figure 10: The settings used for display sACN packets in wireshark

7.4 Version Control: Git, Gitlab, Github

Like most modern software engineering projects a version control system was utilised. Even though there is only a single developer and so the collaboration tools were unused during the duration of the project version control still provides significant advantages related to being able to roll-back versions of code if a change must be reversed. This is particularly useful during the implementation stage of the project where bugs found in testing can be traced to where they were created by testing earlier versions of the code. As-well as the local git repository the code was also pushed onto separate private github and gitlab repositories. This allows development to continue anywhere that can access github and the repository as well as providing two separate backups of the project with one within the school gitlab and one on a private github. The gitlab repository was also useful as it allowed the project supervisor to monitor progress and inspect the project as required.

7.5 Test Coverage Tool

The grcov [39] tool was used to show code test coverage of the library. This tool was created by Mozilla and is made specifically for usage with Rust programs. The output of the code coverage is a webpage which contains details of the lines and functions covered for each part of the library, this is then used to find functionality which has been missed in testing. The

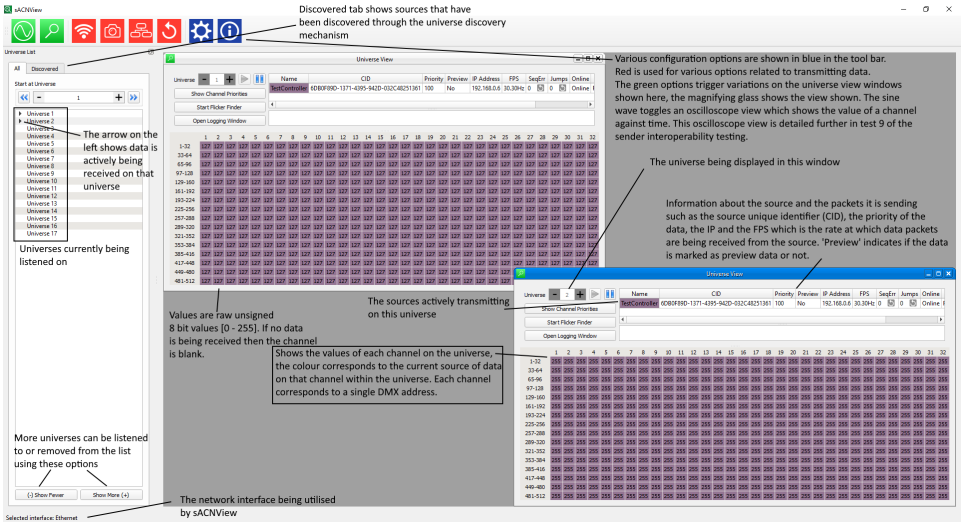


Figure 11: A screenshot from the main part of the SACNView program (v2.1.0) with added explanation of the various features

"usage.pdf" file contains details about how to re-run the coverage tool. It should be noted that the code coverage tool only shows the coverage from the unit and single machine integration tests and so some cases may have been covered in other tests even if not show as covered in the coverage. Grcov version v0.5.13 was used for this project.

7.6 Compliance Testing Tools: sACNView

sACNView [30] is a simple tool which allows sending and receiving with the sACN protocol. It is used as part of the compliance testing of this library as it acts as a real-world deployed version of the protocol which can be tested against. This viewer notably provides support for the universe discovery feature which isn't supported in the other tools used to test compliance making it particularly useful.

The main page for this tool says it is for the ANSI E1.17 [1] protocol (the base ACN protocol) however this is a typo and it really means the ANSI E1.31 protocol (the sACN part of ACN). The reasoning for this assumption is that at multiple points within the documentation such as [**?**] it says things such as 'E1.17 (2018)' which must mean E1.31 (a related part of E1.17) as there is no 2018 version of E1.31. The documentation also describes the universe discovery feature which is not part of E1.17 as it is part of E1.31. Version v2.1.0 of sACNView was used with this project.

7.7 Real-world Usage Tool: Visualisers: Vision

When creating a lighting design for an event a common step for a lighting designer is to create a 3D model of the stage and include in it the planned lighting. This allows the designer, clients and project management to see what is being proposed. Once the design is approved the visualiser acts as an accurate simulation of the behaviour of the lighting fixtures and this allows the lighting programmer for the event to create many of the lighting patterns, effects and cues ahead of the actual fixtures being put in place. This is a massive part of any major project as a week spent using a visualiser to create much of what is required for the event in terms of lighting effects is significantly cheaper than a week doing it with the real-fixtures. It also means that the visualisation can be done without being on site which might not be possible if the site is in-use in the time leading up-to the event. To allow this programming to be transferred straight onto the real lighting setup most visualisers allow usage of the exact same protocols and mechanisms used within real-lights to be used with the visualiser. This makes a visualiser an ideal tool with which to test the created library against as a professional visualiser is designed to simulate real-world usage as closely as possible and contains a professional developed/maintained/tested version of sACN. This means that if the library works with the visualiser it shows that it is conformant with an industry implementation which itself is created to be compliant with the protocol and therefore this provides evidence that the library is compliant. From an outside user perspective it also crucially shows that the library can actually be used for its intended purpose. The visualiser that is used for testing this library is Vectorworks Vision Plus 2019 Version 24.0.6.521266 [32], this software is professional paid software so for this project the St Andrews Students Association's copy of the software + license was used. Permission for this was granted through communication with the current director of events and services as-well as building management. Vision does offer a free-trial version but this has limited features so by using the full-version it helps better show compliance. The relevant parts of the visualiser are explained within the relevant interoperability/acceptance tests but an explanation of the interface is also included in figure 12.

The project extends it thanks to the St Andrews Students Association, particularly the director of events and services (Mika Schmeling) for their permission to use their real-world visualiser setup with this project.

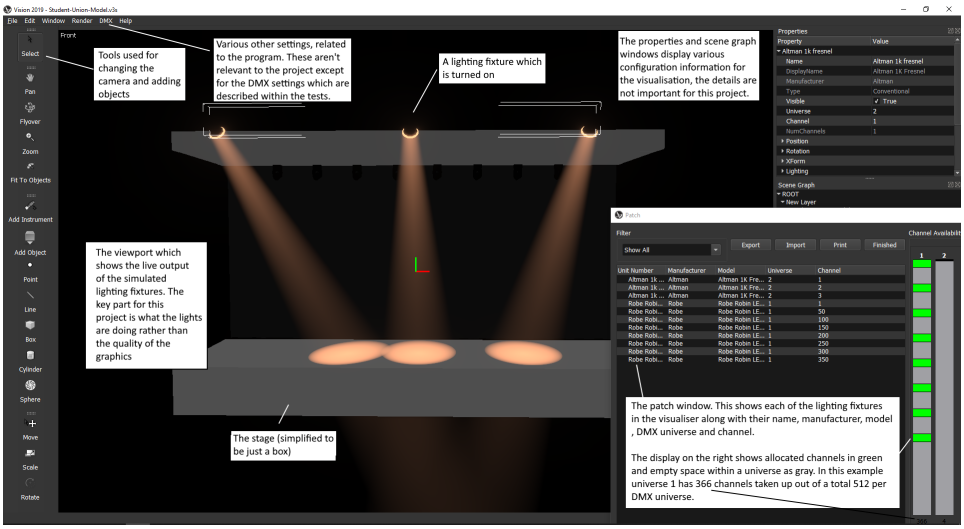


Figure 12: A screenshot from the vision visualiser with added explanation to explain the interface.

7.8 Real-world Usage Tool: Lighting Control: Avolites Titan

The visualiser provides a way to test that the library can send sACN which can be utilised by a real-world system but it doesn't test the other-way around where the library is receiving sACN. To test this a source of sACN was required and this came in the form of a real-world lighting controller. The lighting controller used was an Avolites Titan Mobile [33]. This controller is part of a family of Avolites lighting controllers which are used across the world to control lighting systems with sizes ranging from small few light setups up-to arena sized world-tours. All the different controllers in the family run the same Avolites Titan software. This software is another professional example of a product which claims compliance with sACN and through its extensive usage by professionals in a variety of systems has show that it is conformant with many systems using the sACN protocol. By testing the library receiver against this lighting controller it therefore adds evidence that it is compliant with the protocol and can actually be used for its intended purpose. This controller was available for the project as I already own it for use as part of my work in the lighting industry outside of university. A brief explanation of the Avolites Titan software is included in figure 13, the interface is also further detailed within the relevant interoperability/acceptance tests. Avolites Titan V11.4 was used for testing. Show files (the saved configuration + data Avolites save files) for the tests are included in the interoperability tests folder. Note that each of the folders e.g. "CS4099-Receiver-Interoperability" is a show-file with the files inside all being part of the overall show-file + show-file versions.

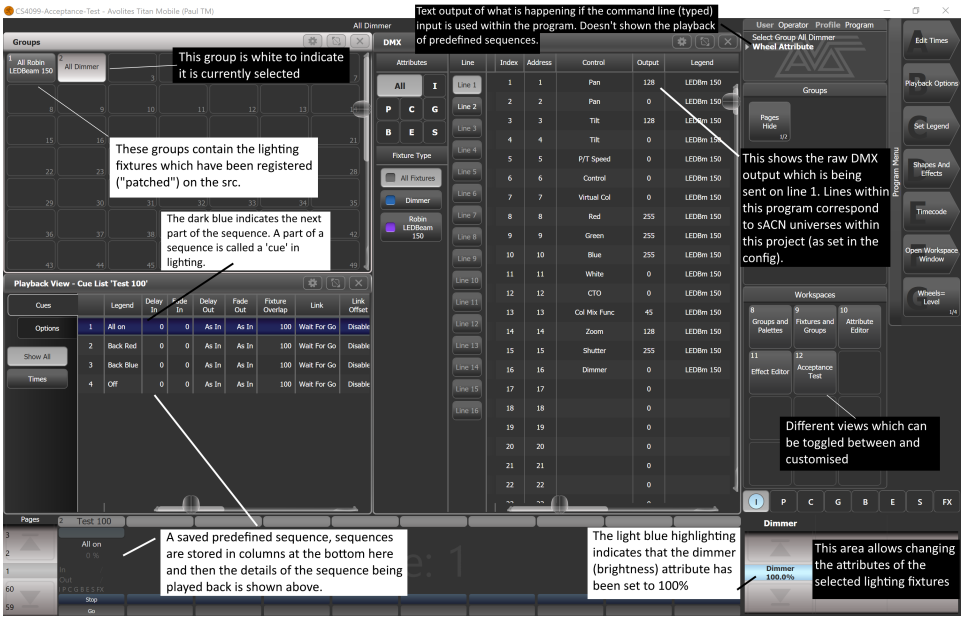


Figure 13: A screenshot from Avolites Titan with some of the features explained.

8 Ethics

This project has no ethical considerations that require notification in this section.

9 Design

The project design is separated into 3 areas. The receiver 'SacnReceiver', the sender 'SacnSource' and the protocol used to communicate. The receiver and sender are completely separate with communication only being performed in one direction from sender to receiver and only using the ANSI E1.31-2018 protocol. This means that each part can be developed completely independently. The design of the protocol used for communication is already established by the protocol specification and described within the context survey section of the report. This leaves the design of the sender and receiver which must be created so as to support the operation of the protocol while also taking into account the non-functional requirements for the project. Figure 14 shows the same example setup shown earlier in the report (Figure 2) but with the areas where the SacnSource and SacnReceiver fit in shown.

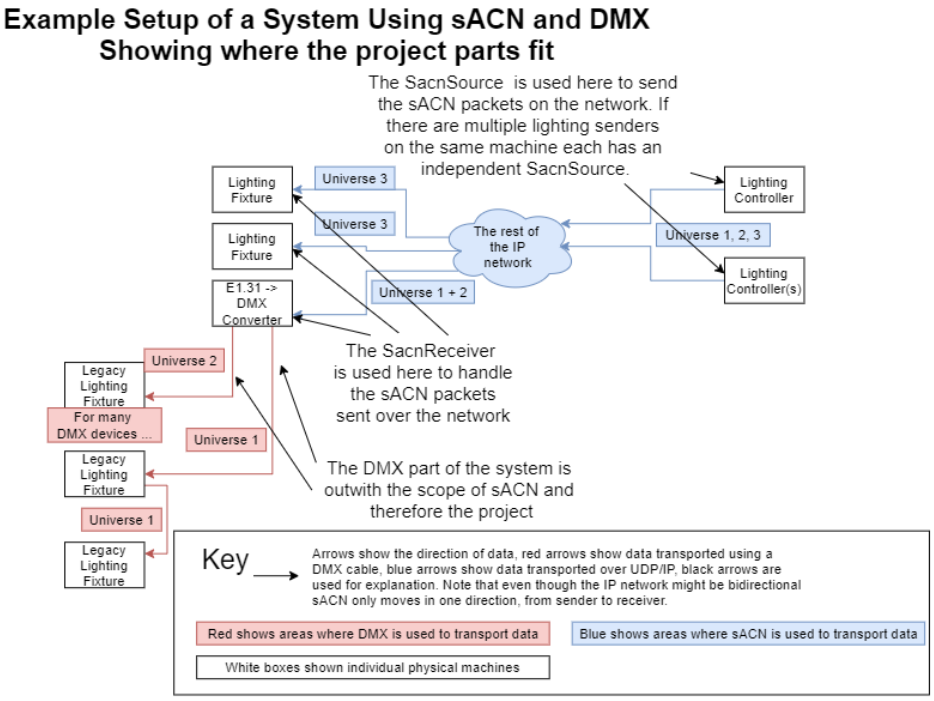


Figure 14: A diagram showing where the 2 separate parts of the project are expected to be utilised within an sACN and DMX system

The design decision was made from the start that the project should take the form of a library which is called from other code to actually perform actions. This means that the program cannot generate data on its own and as part of this it should treat the data as transparently as possible. This simplifies the design of the project as it does not have to specifically handle various cases related to the data transmitted and instead defers this to the program using the library. This is similar to how many other network protocols work such as TCP which just handles its part of the process to move data and doesn't worry about the specific data being sent. The decision was also made that the library should handle much of the sACN specifics as transparently as possible. This means that concepts like universe synchronisation and discovery should be hidden from the user as much as possible to allow developers to focus on actually using the library rather than having to learn exactly how sACN works. Based on the existing implementation the project follows an object orientated type programming design with the SacnSource and SacnReceiver being objects which contain all the methods and state for sending and receiving.

9.1 Sender

The sender can be split up into 4 different areas one for each of the E1.31 packet types (data, synchronisation and discovery) and lastly for stream termination. The data and synchronisation areas are fairly simple on the sender with a data or synchronisation packet being sent when asked by the user with the library handling the formatting of the packet. The universe discovery mechanism is more complicated however as it requires the sender to send discovery adverts on an interval and not just when the user is sending data.

To address that the sender must send universe discovery packets on an interval requires that the library is able to run periodically. There were 2 possible mechanisms considered for this which were decided between. The first is to require the user to periodically call the library using a poll function. This allows the library to perform any checks it requires and send packets when needed. The problem with this however is that it means the user must keep calling the library and they must do so often enough that the packets will be sent within the required universe discovery interval. This mechanism works well for the receiver were the receiver can sit in the background and only needs to update when the user wishes to receive. This does not work as well for the sender however as it makes the library more complicated to use. It does provide some advantages as it allows the user full control over the event loop and when code is run which is useful for some embedded applications but this mechanism was decided against as this was deemed not worth the cost of the added complexity on the user. The second mechanism which was the one chosen is based on having a poll thread. This thread runs in the background when the source starts up and periodically polls the sender. This allows the sender to then send the discovery packets as needed. This mechanism has disadvantages related to requiring another thread to be running which can lead to issues related to concurrent access such as deadlocks and data races. That noted these disadvantages are fairly minor due to the initial design decision to use Rust which already guarantees thread safety and while it cannot prevent deadlocks it guards against many of the other possible issues such as data races [44]. Having a separate thread (or other similar mechanism) was therefore chosen as it allows the sender to handle sending universe discovery adverts without adding complexity to how the library is used.

When it comes to stream termination a key sender design decision is when to send termination packets. One mechanism would be for the sender to timeout each universe itself and if the universe wasn't being used by the user for a period automatically send the stream termination packets. This has the potential for the sender to make the system more efficient because if it knows (for example in a predefined sequence) that a universe won't be used

for awhile it can send stream termination packets to speed up the timeout on the receiver. The problem with this mechanism is that it is significantly difficult to implement for the general case. Therefore the alternative decision was made where the user manually tells the implementation when it is done with a universe. This mechanism allows the user freedom to stop using a universe whenever but without forcing them to as the receiver can just timeout the universe as normal. This puts all the work in terms of the required data-loss timeouts onto the receiver which makes sense given that it already has to handle this. This solution does mean that a universe might be marked as sending by a sender for longer than it actually is but this isn't a large problem as this doesn't mean more data is being sent and so will only have the effect of a small delay (2.5s timeout on receiver as defined in [3] Appendix A).

9.2 Receiver

The receiver can be further split up into how it handles 2 areas of the protocol. The first is how it handles data and universe synchronisation. The second is how it handles universe discovery. While in practice these may be combined closely together at the design stage these 2 areas can be treated separately with how they are brought together being an implementation detail.

9.2.1 Data and Universe Synchronisation

The data and synchronisation mechanisms are closely linked as both effect how data is handled by the receiver. The mechanism for how the receiver should behave both with data and in a scenario involving universe synchronisation is detailed in the "Sync-Mechanism.pdf" file. This file shows that the receiver must have a way of storing packets for later and then triggering them at the correct time.

9.2.2 Universe Discovery

The universe discovery mechanism doesn't directly relate to the way data is handled and therefore can be treated separately to the synchronisation and data part of the receiver. The expected behaviour of the receiver is detailed in the "Discovery-Mechanism.pdf" file. This highlights a key design decision which was to abstract over the sACN specific behaviour on receiving a universe discovery packet such as rebuilding the universe list from multiple pages. This makes the mechanism easier for a user as they don't have to worry about the sACN specifics on how to handle the individual discovery packets and instead can just make use of the functionality.

10 Implementation

As noted within the design the library can be divided up into 3 sections, the receiver, sender and the protocol between. Within the implementation this corresponds to the 3 files, receive.rs, source.rs, packet.rs. receive.rs contains the SacnReceiver struct which is used for all parts of the library related to an sACN receiver. source.rs contains the SacnSource struct which is used for all parts of the library related to an sACN source. packet.rs contains the parts of the library related to sACN on the network and so contains the parsing and packing implementation as-well as functions which are utilised by the receiver and sender such as the function to convert a universe to a multicast address. Packet.rs also contains all the sACN constants.

10.1 SacnReceiver

As part of developing the receiver an implementation decision had to be made about how the receiver should interface with the users code. One mechanism which was considered was to create an event-based call back system, this is like how many JavaScript libraries are created and is based around the receiver being asynchronous as a call-back could be called at any point. This wasn't chosen as it makes the library more complicated, this increases the chance of bugs and to add to this there is no standard 'Rust' way of doing callbacks. This means that there was far more chance of the library being unstable or liable to fail in certain situations as these callbacks add extra concurrency considerations due to requiring a constantly running receiver thread to trigger the callbacks. It is also problematic for some embedded devices which can struggle with the concept of being continuously interrupted to perform an action especially if they don't finish the action before the next occurs. The mechanism that was chosen therefore was to base the majority of the receiver functionality around an easy to use 'recv()' method. This method takes a timeout and blocks until either data is received or the timeout is reached, this is very similar to a standard UDP or TCP socket and so the functionality should be familiar to many developers already. The recv method also allows a timeout of 'None' which means it will block for an indefinite amount of time until there is data ready to pass up. Something that is different with this recv() compared to say a UDP socket however is that the universes to listen to must first be specified. Rather than having multiple instances of an SacnReceiver per universe the implementation allows many different universes to be received by a single instance. Besides being more efficient this makes it much easier to manage for the end user but it does still require the user to specify the universes they want to receive ahead of time.

If a user wants to receive data then once they have listened to the uni-

verses they want they can call recv and the code will block until data that is ready is received. A key point here is 'data that is ready'. The recv() implementation completely abstracts over the concept of universe synchronisation on the receiver side, if a synchronised data packet is received it won't be passed up to the user library until the corresponding synchronisation packet is received. This makes handling synchronisation significantly easier for the user based on the observation that the user shouldn't be acting upon synchronised data that hasn't been synchronised as per the protocol and so therefore there is no reason to make them handle it. In addition to this universe discovery and termination are also abstracted over for a similar reason. This means that for a user to get up and running with a receiver that can handle receiving as many universes as required (and that the hardware can handle) with support for synchronisation is very quick.

The user may still want to utilise the universe discovery feature to check the discovered sources so this is still possible by using the 'get_discovered_sources()' method which returns a list of sources that have been discovered. This also made to be as simple for the user as possible by abstracting over the concept of timeouts and pages by automatically timing out sources when retrieving and presenting the universe lists as complete lists with no page divisions. This means that the user never has to understand about the page fragmentation performed by the protocol and can just utilise the feature for its functionality. This is similar to how many link-layer protocols will automatically fragment and then re-build big packets so that they can be sent over a link which has a limited maximum transmission unit e.g. Ethernet's 1500 Byte MTU.

There are still situations where a user might not want some events to handled silently. For example if the library is used within a receiver which needs to perform an action whenever a new source is discovered on the network. Therefore to support this the library allows setting an 'announce_source_discovery' flag. This utilises the existing 'Result' return type to return a special error if a new source universe list is successfully discovered through universe discovery. By utilising the existing system this allows this option to be easily toggled on and off without it effecting the handling of the recv() method by the user. This is a behaviour very similar to that used for timeouts when receiving from a UDP socket. Using the same mechanism the library also allows stream termination packets and timeouts to be announced. This allows the user a greater level of flexibility without complicating the basic case. The decision not to make announcement the default behaviour was based on the assumption that in most use cases the receiver spends most of its time receiving and processing data rather than for example listening for more discovered sources - given most sources will probably send more than a single data packet there are significantly more

data packets than sources to discover so most receivers spend most of their time receiving data.

One potential trap of this mechanism is attempting to receive with no timeout, all announcements disabled and no universes being listened to. This would lead to the recv never being able to return successfully. To help protect the user from this happening the implementation will return an error if the user attempts to call recv() in this situation.

A single instance of an SacnReceiver only allows receiving using IPv4 or IPv6 at one time. This means that if a user wishes to utilise both IP protocols simultaneously they must create 2 SacnReceiver instances. It is expected that is a fairly rare use-case which is why the library does not provide explicit built in support for this.

Within the SacnReceiver the actual interfacing with the IP network is further abstracted by the 'SacnNetworkReceiver' receiver. This wraps around the UDP socket that the sACN packets are actually received on to allow receiving to be done in a way that is independent of the actual socket used. The reason for this is that it was found after much trial and error that the socket provided by the underlying rust Socket2 library used cannot be used identically on both windows and linux. This meant that 2 different implementations of SacnNetworkReceiver were created with the version selected based on the targetted compilation operating system. This allows the user using the sACN library to program without having to worry about the operating system (as long as its either Windows or Linux based) as the implementation has handled this. One limitation related to this was also discovered which is that within rust there currently (at the time of writing) does not exist support for receiving UDP over IPv6 multicast on windows with multiple multicast groups on the same socket within any of the libraries tried (std, Net2 [45], Socket2 [46]). This does provide some limitation on the usage of the library however it was beyond the scope of this project to attempt to implement and full test an implementation of IPv6 multicast for windows. Due to the separation in the receiver between the part that interfaces with the network and the rest of the code it does mean that if/when this is implemented in rust it can be easily added into the library without having to effect the rest of the code base or change any public facing functionality.

10.2 SacnSource

The sender side of the library is split up internally between the main functionality and the universe discovery poll thread. The main functionality

is encapsulated within the 'SacnSourceInternal' struct which is wrapped within the SacnSource. This actually handles all the sending of sACN data including the universe discovery packets. All the thread does is periodically wake, check if it is time to send a discovery packet (10 second discovery packet interval defined for sACN in ANSI E1.31-2018) and if so calls the method on the SacnSourceInternal and then goes back to sleep. This keeps the discovery thread as simple and lightweight as possible and allows all the functionality to be in one place that is easier to test.

The SacnSourceInternal is encapsulated within a rust Mutex which is itself encapsulated in a rust 'Arc' . The reason for the mutex is that because the receiver is made up of 2 threads there is the possibility that one thread could interleave while another operation is in progress. This can lead to data-races which could leave the sender in an inconsistent state. The usage of a mutex therefore prevents this by only allowing one thread to perform an operation on the sender at any one time. The decision was made to make this a coarse grained lock with the Mutex covering the entire SacnSourceInternal. This does mean that the discovery thread or user-called-methods may have to wait on each other for slightly longer but at the advantage of not requiring a more complex locking system. This makes bugs less likely and especially since the use of another thread is only to perform relatively infrequent and short tasks the added costs in time and required testing to implement more fine-grained locking were not deemed worth it. The second layer of encapsulation is within an 'Arc' which refers to 'Atomically Reference Counted' [47]. This is a rust reference counting implementation which is thread-safe through the use of atomic operations. This means that it allows multiple different concurrent threads to reference the contained mutex simultaneously, in this case the application and universe discovery threads. This prevents the problems which can be related to this such as one thread freeing the memory for an object while another thread still has a reference to it. Due to the reference checking present within Rust even if you attempted to not use an Arc+Mutex the compiler would prevent this as it violates rusts thread-safety policy. This is mostly transparent to a user of the library except that the rust error system requires that the possibility of a 'Poisoned' mutex is handled. This is where a thread which is currently holding a lock on a mutex crashes and is unable to release it. This leaves the contained structure in a potentially inconsistent state and so it cannot be reused. The implementation handles this case by returning a SourceCorrupt user to the error. This is not expected within normal operation but is still possible. It is expected that in most cases if a user did encounter this error they would handle it by creating a new instance of the SacnSender and continuing operations. The library could have attempted to handle this transparently but this was decided against in-case the user is using the library in an environment where the standard mechanism of just creating a

new instance isn't suitable (for example in some embedded devices).

Like the receiver the sender attempts to make the most basic use cases as easy for a user to get to grips with as possible. To this end there are 2 methods needed to allow sending data once an SacnSource has been created. The first is to register the universe, this is required ahead of time as part of the universe discovery mechanism because it allows the source to add that universe to its advertised list of universes. Once a universe is registered the second is the 'send()' method. This method allows a user to send data to 1 or more universes with a given priority, destination IP and synchronisation address. In the most basic case of sending unsynchronised data at the default priority using IP multicast the last 3 options can all be replaced with 'None' with just the data and destination universe(s) provided. This allows a user to start with a simple base which can be configured for more applications as required.

Unlike on the receiver on the sender side universe synchronisation is handled explicitly. The reason for this is that it allows a greater level of control by the user while not adding to much complexity. An early implementation did attempt to have synchronisation packets sent automatically after a synchronised data packet but this meant that the library could only be used in the situation where the data packets send should be synchronised immediately and didn't allow the user to delay it or wait for some other option. The way the user interacts with the universe synchronisation feature is by providing an synchronisation universe as an argument to data send using the send method or None if no synchronisation is desired. The user can then trigger the synchronisation by using the 'send_sync_packet' method which sends a synchronisation packet to the given synchronisation address. This allows a much higher level of control by the user on when to synchronise data and allows for the possibility that the synchronisation might be performed by another sender. While not a hard requirement it is advised in the standard (ANSI-E1.31-2018 Appendix B.1) that there is a small delay between sending data and sending the synchronisation packet to allow receivers time to process the data. This isn't enforced by the library as what counts as a 'small' delay will depend on the system and so this is left up to the user to decide. Similarly as specified in ANSI E1.31-2018 Section 6.6.1 the send method shouldn't be called at a higher refresh rate than specified in DMX (ANSI E1.11) unless there are no E1.31-DMX converters on the network. Since this is also something which is system dependent and the library cannot know on its own this is also left to the user to control how often they call the send method

The sender supports all 3 IP modes specified in the non-functional requirements of unicast, multicast and broadcast. The default mode used is

multicast and this is accessed by the user providing 'None' in the place of the destination IP in the send (or send_sync_packet) method. This will cause the sender to use the destination universe (or synchronisation address for sync packets) to find the multicast address to use. Unicast can be accessed by instead providing the destination IP. This will cause the source to send the packet using unicast. If broadcast is desired then the user can provide a broadcast IP destination address and this will cause the library to send the data to that broadcast IP and thereby broadcast the data to all receivers on that subnet. The ANSI E1.31-2018 protocol specification [3] only discusses universe discovery in the context of IP multicast so therefore this feature only utilises that sending mechanism. It is relatively easy to modify the library later if required to also allow sending discovery packets using other IP modes.

In some situations it might be required that universe discovery isn't used, for example if there are devices which implement ANSI E1.31-2009 which was created before universe discovery and which don't correctly discard packets with the wrong vector. To allow compatibility with these devices the sender provides the 'is_sending_discovery' flag which defaults to true but can be set to false to prevent discovery packets being sent.

10.3 Protocol Packet Parsing / Packing

The parsing and packing of sACN packets is handled by the packet.rs file. By keeping packing and parsing together it makes it easier for both sides of the library to be kept consistent as a developer can make changes to both areas in a single place knowing that this is the only areas that need changing. Within the file it is separated by layer with each layer parsed as a separate struct. The macro system used for defining the functions at each layer is from the existing implementation and kept as it provides essentially the same functionality as doing it the 'standard' way as shown in the other files. A large amount of the parsing and packing code was brought over from the existing implementation but a number of changes were made. For example the replacing of the old rust style error system with the new library error-chain system with a specific type of error to encapsulate all parse/pack related errors. Related to this a large amount of error checking that was missing was also added and the code was extensively tested to find problems. In-doing so a number of bugs such as the data packet options field being packed incorrectly were found and fixed. In that specific example the original implementation was referencing bit 7 in the option field to mean the 7th bit when in the ANSI E1.31-2018 specification the bits were 0 indexed so bit 7th was actually the 8th. This lead to all the stream termination and preview packets produced by the implementation to be incorrectly utilised by external receivers. A level of code tidying on the existing code was also

performed with magic numbers replaced with descriptive constants and the code documentation extended.

One implementation decision that had to be made was related to the ordering of the universes within a discovery packet. ANSI E1.31-2018 Section 8.5 specifies that the list of universes in a universe discovery packet must be sorted numerically but doesn't specify if this should be in ascending or descending order. This meant the implementation had to make a decision on which to use with the assumption being to use ascending order however this exists as a potential source of compatibility problems between implementations due to this being unclear in the specification.

10.4 Std vs Non-Std

The library is implemented assuming a std environment. This means that the rust std libraries such as the network library are available. This greatly increases the amount that can be done using rust as within the standard library the inbuilt functionality is fairly limited and would require rebuilding many already implemented solutions. This differs from the library that the implementation is based on which allowed running in environments with and without std. The reason to discontinue support for no_std environments was made as the new parts of the protocol which were being implemented such as universe discovery are significantly easier and provide a better user experience when parts of the standard library such as the threads can be used.

10.5 Drop / Closing / Termination

Unlike many languages the underlying rust socket used for this library does not require being explicitly closed as it automatically cleans up when it is out of scope. This protects against an entire family of errors related to incorrectly closed/not-closed streams. In-line with this the library also cleans the created receiver/sender up automatically through the implementation of the 'Drop' trait. As part of this the implementation decision was made that a source will automatically terminate any universe it is currently sending on (as well as its spawned thread), this is similar to a TCP stream sending FIN packets to close. The receiver will also de-register any universes it is listening on by leaving the multicast groups it has previously joined.

During a Drop there is no provided way to pass an error to the user. This is inherent in Drop being called automatically because there is no clear return path for an error nor can the Drop be stopped either-way as it might be called during a panic! or program wrap up. This leads to an imple-

mentation decision for how to handle errors with 4 distinct options: panic, notify, prevent, ignore. Panic means that within the drop if there is an error then panic the program with a description of the error. This notifies the user however it also causes the program to be terminated even if it could potentially continue. It also pollutes the error-output and can lead to the original error/problem to be hidden. This makes it unsuitable for this application as a failure to drop won't lead to memory unsafety or otherwise cause significant problems beyond the program scope. Notifying the user refers to using a functionality such as logging, printing to standard-out or some other mechanism. Printing to stdout is avoided as this might pollute an applications output by displaying errors to the user which a developer using the library might want to avoid. Logging is a possible option which adds further complexity to the library as-well as another requirement for a logging library. Preventing the possibility of the error is the ideal solution however in this case as IO is required this cannot be guaranteed if there is a problem with the underlying socket. The final solution is to ignore the error, usually this would be problematic as it doesn't allow the application developer to decide the programs behaviour or to fix the problem but in this case that isn't possible anyway. Ignoring the error also allows as much to be cleaned up as possible unlike panic which would stop cleaning up at the point the error occurs. The actual implementation decision based on this was therefore to ignore the error with the recommendation that logging could be potentially added later.

If the user wishes to they can also terminate a specific universe manually. This is done through the 'terminate_stream' method on SacnSource. This will remove the universe from the universe discovery list adverts for the source as-well as send the termination packets.

10.6 Errors

The base implementation provided its own error system based on an Enum with various different types. This had a number of problems, the biggest two being that it didn't allow errors to be encapsulated within each-other to provide a back-trace and it wasn't compatible with errors from rust libraries such as Io and Net. Since this error system was created for the existing implementation (before or during 2018) there has been significant changes within rust and the way that errors are handled. For example the 'try!' [27] macro which used to return if the item produced an error type has since been depreciated, replaced with the '?' operator and 'try' made into a reserved word.

Theses issues meant that the existing error implementation was no longer suitable and it didn't make sense to continue trying to use multiple errors

systems (rust Io/Net, old system, new errors added by new features). Therefore the entire error system was replaced using the Error-Chain library [28]. This library is frequently used throughout the rust eco-sphere and allows combining all the error systems into one system with rust errors automatically converted as needed. It allows errors to be encapsulated within each other which allows chaining of errors together to produce much more informative back-traces. As part of this update of the error system all usages of the depreciated 'try!' macro were removed and replaced with the new '?' operator in combination with the error-chain 'bail!' macro. A number of new errors were also added to more descriptively describe possible errors within the library as listed in the 'error.rs' file such as 'ExceededUniverseCapacity' and 'DmxMergeError'. To aid usability all errors related to the parsing and packing of sACN packets were moved into their own 'SacnParsePackError' error-chain. This allows a user of the library to handle all these errors by just handling the generic SacnParsePackError which is useful if they don't care specifically why a packet was malformed just that it was. If the user does want to handle specific parse-pack related errors they can also do that in the usual way by matching against SacnParsePacketError(x) and then checking the ErrorKind of x which will be one of the parse-pack errors. Examples of this are shown throughout the unit tests for example test "test_malformed_data_packet_framing_layer_wrong_vector_parse" in the "data_parse_tests.rs" file demonstates this in use to check that the expected error is returned.

Programs which utilise the library are not required to continue using error-chain within their code and can use their own error systems however for the 'demo_src' and 'demo_rcv' programs the decision was made that continuing to use error-chain made sense due to the advantages it provides as described above.

11 Testing

As a software engineering project the testing stage is vitally important and took up a large amount of the total time spent on the project. The key aims of testing are as follows, first to show that the code works as intended. This is primary done through the unit and integration tests. The next part is showing that the code works as expected by the protocol. This is referred to as compliance testing and as creating a library that is compliant with parts of the ANSI E1.31-2018 protocol was the main aim of the project this compliance testing is crucial to showing the success of the project. The final aim of the testing is to show that the library is actually suitable for usage, this comes in the form of acceptance testing where the library is actually utilised for its intended purpose by an end user.

These 3 aims show that the project works as intended, that the intended functionality is compliant with the protocol and that the project can actually be used for its functionality by a user. Once the project has successfully passed these 3 categories of tests it is ready to move onto the deployment stage of the software development life cycle.

11.1 Scope

Within these testing objectives 3 types of tests were used Normal, Extreme and Exceptional. Normal testing involves situations which are expected by the project such as in this case receiving data from a registered and expected universe.The basic requirement of the project to be successful is for it to pass all normal testing as this shows that the project actually performs the intended functionality. Extreme testing is similar to normal testing but with the situations being on the edge of what is expected/allowed, for example sending a full data packet is an extreme test as it is on the edge of the allowed data packet length. Extreme tests show the bounds of the project and highlight where normal, expected scenarios transition into exceptional scenarios. Exceptional tests are where the project is tested with scenarios and inputs beyond what is allowed. In these cases the program must take some action to handle the scenario, in many cases exceptional input may put the program into an undefined or failing state. This action is undesirable however because it means that there is no way to know exactly what will happen or in some cases to stop the program from crashing. This is especially problematic for this project as it is a library which should be usable by developers within their projects and different usages of the library will need to handle errors in different ways. The project therefore attempts to prevent undefined or crashing behaviour by flagging up exceptional input or scenarios before they cause a crash (called a panic in rust). This flagging is done by methods/functions being able to return a Result type which explicitly encodes the possibility of either an Ok' non error result or an 'Err' error result. The aim of the exceptional tests therefore are to show that even when provided with unexpected scenarios such as those out-with the ranges given by the protocol that the implementation follows predetermined behaviour that allows the user of the library to handle or correct the problem.

The testing aims to provide coverage of the entire library including the parts from the existing implementation. This is important because the existing implementation lacked sufficient testing and in multiple cases it was discovered that it had problems including multiple deviations from the protocol specification. One example of this was the 'options' field within the data-packets. The existing implementation took bit 7 to mean the 7th bit which is incorrect as specified in ANSI E1.31-2018 that the 7th bit means

the most significant bit and the 0th bit is the least significant. This problem meant that the existing implementation did not correctly assign the option flags leading to malformed packets and this is a problem which was prevented through testing. In this case the problem was verified to be fixed both through unit testing and by utilising wireshark to verify that it was interpreting the option field as expected once the change was made. This highlights why thorough testing is so important as small differences which have a big impact such as this are easy to miss when developing and unlikely to be spotted in a developer 'dry-run'.

11.2 Testing Mechanisms

A priority was put on reproducibility and automation when it comes to testing. The reasoning for this is once a framework is set-up it takes approximately the same time to run a test manually once or twice as it does to write the test in a way that it can be run multiple times automatically. This means that there is only a small penalty to setting up a test so that it can run automatically but once it is set-up it can be run frequently allowing confidence that the code continues to work and that any change such as a bug fix for another test hasn't broken something else. Easily reproducible automated tests also provide a significant advantage to a project once it reaches the deployment/maintenance stage as they act as further documentation of the code and a source of examples for new developers to use when learning. These examples are particularly good as they can be run to verify that they still perform as expected which can be used to flag up areas where the documentation and code have diverged.

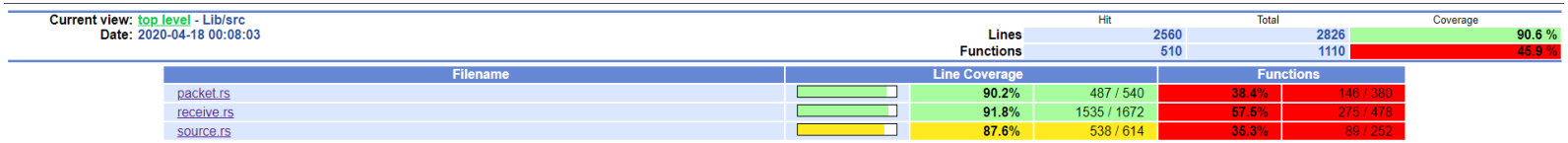
11.2.1 Unit Testing

Unit tests focus on a small specific part of the program to test its functionality for example a single method or functionality. The goal of these tests are to be quick to run and show that each individual part of the project work. By having unit tests for each part of the program is allows showing that individually all the bits of the project work. As these are quick to run they are run after each change/bug fix made during the testing stage of development and show that the fix hasn't introduced any issues into already working sections. This sped up the implementation and testing phases by allowing problems to be identified and fixed quickly. These tests are also an important part of the maintenance stage for a similar reason. The output from the unit tests is included in the "unit_test_results.txt" file which shows that all unit tests and example code passed as expected.

Unit testing relies on testing each unit to be effective and so therefore as

part of testing a code-coverage tool was used. This goes through the code and highlights areas that are missing tests thereby making it easier during testing to identify missed areas. It isn't perfect as it cannot check if every possibility/situation has been tested for every function but acts as a guide to improve testing.

As described in the tools section unit tests created using the in-built rust/cargo unit testing framework. In addition to this the code coverage of these unit tests was checked using the grcov tool created by mozilla [39]. The output of the test coverage is included as a webpage (index.html in the coverage folder) and the library view of the results are shown in 15. The library is focused on as this is the focus of the testing (the other modules are part of the webpage but not shown in the screenshot). The code coverage tool shows that the test provide good line coverage with packet and receive particularly well tested. This makes sense as a large amount of testing went into the packet parsing specifically as part of the compliance testing. The low scores for the functions column would indicate poor coverage but actually inspecting the coverage for each file shows that the majority of functions are covered. There is limited documentation available for the grcov tool so it is hard to say why the functions score is so low while the line coverage is high. One possibility is that in a few functions there are situations which were not tested. This includes possibilities that are extremely difficult to trigger intentionally such as the PoisonedMutex error discussed previously. An inspection of the files with coverage highlighting also indicates multiple non-code areas being highlighted which potentially indicates a problem with the code-coverage tool used.



Current view: top level - Lib/src
Date: 2020-04-18 00:08:03

	Hit	Total	Coverage
Lines	2560	2826	90.6 %
Functions	510	1110	45.9 %

Filename	Line Coverage			Functions	
packet.rs		90.2%	487 / 540	38.4%	146 / 380
receive.rs		91.8%	1535 / 1672	57.5%	275 / 478
source.rs		87.6%	538 / 614	35.3%	89 / 252

Figure 15: The library view of the code coverage tool output showing the code coverage of each part of the code, more detail is shown in the include coverage webpage / folder

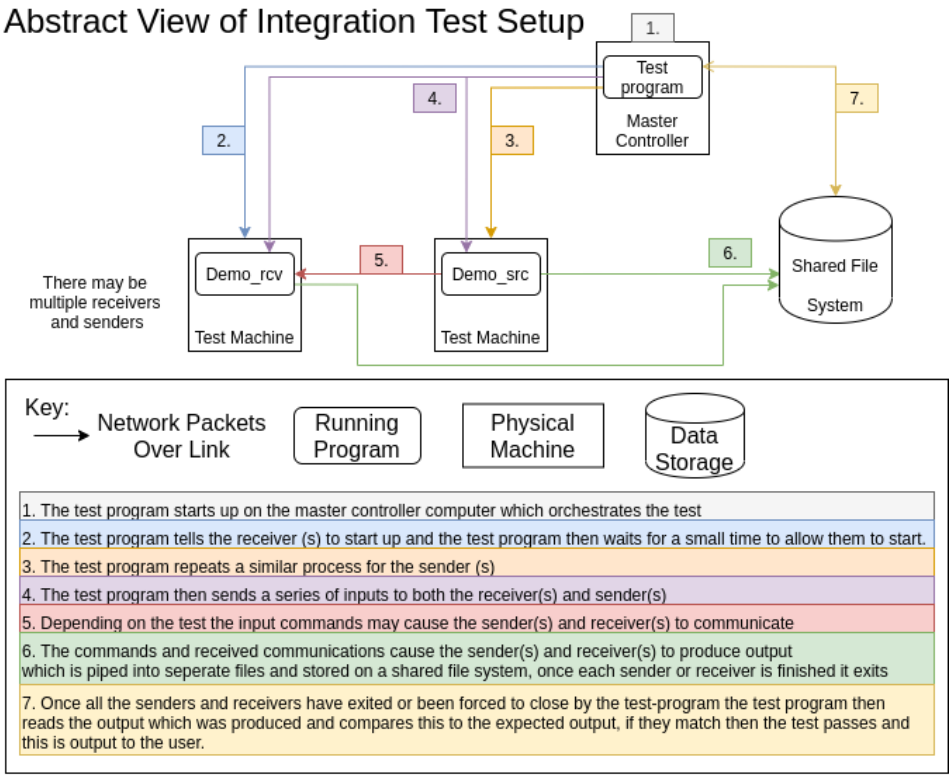
11.2.2 Integration Testing

Unit testing shows that individual parts of the code works as expected, integration comes as the next step where the parts are tested (integrated) together. This included testing the sender and receiver as full units as well as testing them with each other. This was done using two separate mechanisms. The first utilised the rust testing framework that already exists (as used for the unit tests) by creating multiple threads within a test with each thread representing a sender or receiver. These senders and receivers then connect to the same network using different addresses and generate/receive packets to check the output in a variety of scenarios. This simulates the senders and receivers being independent as they only share data through the protocol with no shared memory (sharing memory between threads can only be done explicitly in rust a normal variable cannot be shared). The only exception to this was the usage of the rust thread message passing system which allowed the sender/receiver threads to wait on each other as appropriate to allow repeatedly creating the desired test scenario. An example of this is shown in figure 16. This shows how a typical test is set-up . First the constant parameters are defined first. Second the senders or in this case receiver created are created. The sender/receivers are then put into the expected states by using the thread message passing to communicate as certain points are reached in the code which in this case is the receiver being ready to receive. The actions being tested, in this case sending a single universe of data over multicast, are then performed and the outcome is checked against the expected results. These integration tests mimic the real-usage of the system but with the advantage that the states of the sender/receive can be more easily synchronised to test a specific scenario using the thread message passing system. These tests can also be run on a single machine utilising a feature within both Fedora and Windows which allows a single interface to a network to use multiple IP addresses. This is required because the receiver must use the protocol defined ACN port and so different addresses have to be used to provide separation. As the states can be easily synchronised and only a single computer is needed these tests allow a large range of possible scenarios and functionality to be checked without requiring a more complicated (and prone to breakages) set-up.

```rust
fn test_send_recv_single_universe_multicast_ipv4(){
    // The universe and priority of the data used in this test.
    const UNIVERSE: u16 = 1;
    const PRIORITY: u8 = 100;

    // Allows control of the receiver and sender so that they can be put into the correct state for the test.
    let (tx, rx): (Sender<Result<Vec<DMXData>>>, Receiver<Result<Vec<DMXData>>>) = mpsc::channel();
    let thread_tx = tx.clone();

    // A simulated receiver, this is independent from the sender (apart from the communication channel for syncing states).
    let rcv_thread = thread::spawn(move || {
        // The receiver binds to a test IP and the ACN port. This port is the ported used for this protocol so the receiver must bind to it.
        let mut dmx_recv = SacnReceiver::with_ip(SocketAddr::new(IpAddr::V4(TEST_NETWORK_INTERFACE_IPV4[0].parse().unwrap()), ACN_SDT_MULTICAST_PORT), None).unwrap();

        dmx_recv.listen_universes(&[UNIVERSE]).unwrap();

        // A control message is sent now that the receiver is ready so that the sender can progress.
        thread_tx.send(Ok(Vec::new())).unwrap();

        // The receiver then waits until it receives the data.
        let result = dmx_recv.recv(None);

        // The result of the receiver is then sent back to the original test thread using the control channel.
        // This allows the checking of the results to be done on the first test thread (having the assertions on the same thread behaves better with debug output).
        thread_tx.send(result).unwrap();
    });

    // Blocks until the receiver says it is ready. This stops the sender sending before the receiver is created meaning it would miss the data.
    rx.recv().unwrap().unwrap();

    // The sender is bound to an interface on the same network as the receiver but on a different port.
    let ip: SocketAddr = SocketAddr::new(IpAddr::V4(TEST_NETWORK_INTERFACE_IPV4[1].parse().unwrap()), ACN_SDT_MULTICAST_PORT + 1);
    let mut src = SacnSource::with_ip("Source", ip).unwrap();

    // The sender registers the universe for sending and then sends some test data.
    src.register_universe(UNIVERSE).unwrap();
    src.send(&[UNIVERSE], &TEST_DATA_SINGLE_UNIVERSE, Some(PRIORITY), None, None).unwrap();

    // The data that the receiver received is sent back using the thread message passing channel.
    let received_result: Result<Vec<DMXData>> = rx.recv().unwrap();
    rcv_thread.join().unwrap();

    // Check that the receiver received the data without error.
    assert!(!received_result.is_err(), "Failed: Error when receiving data");
```

Figure 16: An example of the code used for a single machine integration test

Across Machines The threaded integration tests are limited because the protocol is designed to work across multiple machines and the tests only use one machine. This means that to fully test the protocol it also has to be tested in a more representative environment of its actual usage. In-order to allow this 2 small demo programs were created, these programs were also written in rust and represent an example implementation of a sender ('demo_src') and a receiver ('demo_rcv') that uses the library. A testing framework was then created as shown in the 'script-testing' folder. This framework works between multiple machines by using SSH to start up the required senders and receivers and then predefined input is provided to both and the output written to a file on a shared file system. Once all the tests are run the output is then compared against the expected output (utilising the diff tool) and if it matches the test is marked as passed (failed otherwise). This allows a way of showing the that the protocol works across a real network setup with multiple machines while still being reproducible without having large amounts of manual input. This test setup can be represented as the abstract and physical layout shown in Figure 17.



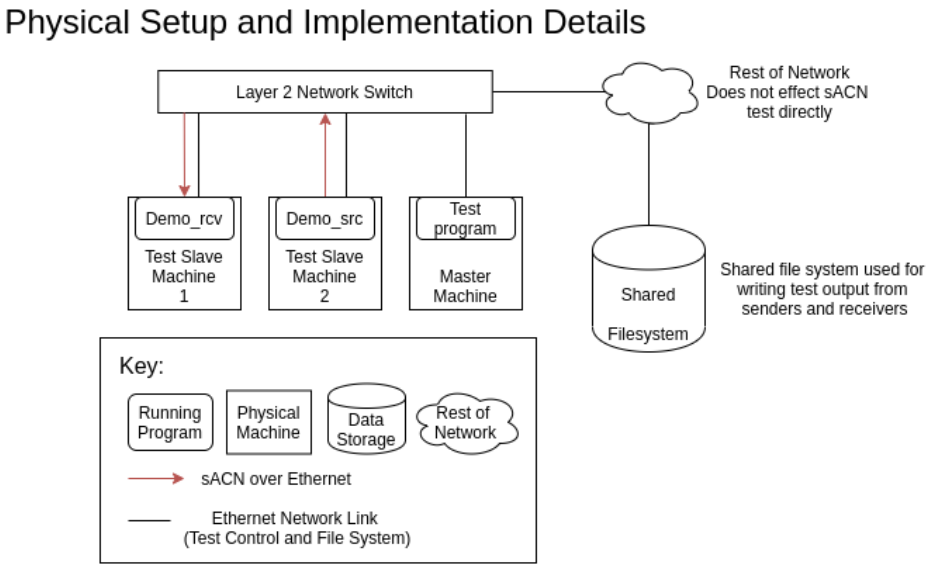


Figure 17: The abstract setup of the integration tests along with the actual implementation

The test-scripts used as-well as the expected output files and given inputs are included in the script-testing/single-rcv-src folder. The test_run script

actually runs the test, it starts the receiver and sender and then provides the receiver the commands in the 'rcv' file and the sender the commands in the 'src' file. The output of both sender and receiver are piped into an output file 'xrcv-out.temp' and 'xsrc-out.temp' where 'x' is the test number. The test_check script then checks the output against the expected output for each test and reports pass/fail. The reason for separate run/check scripts is that it helps to account for the delay in the file-system syncing the output from the test machines so that the master can check it. The test.sh file calls each test / check in turn and is the entry point for the testing mechanism (as described in usage).

More than 2 machines The script above allows testing the features of data sending/receiving, synchronisation, discovery etc. however it only tests between 2 machines. This makes it easier to check the output as by setting the appropriate wait points a loose ordering can be enforced so that the output from the receiver is always the same. This is much more difficult to do however with more than 2 machines. For this reason another set of integration tests was created in the 'script-testing/multiple-rcv-src' folder. These tests utilise the same setup as the 2 machine tests above but allow testing between more than 2 machines. The mechanism used is the same but instead each sender and receiver is given a different input file as indicated by the naming within the test folder so for example the first receiver takes the input from the file 'rcv_1' within its test folder and the second receiver takes 'rcv_2' and so on. Due to the ordering being unpredictable it means that the exact output from the receiver is hard to know ahead of time. Knowing what is expected in general terms (e.g. 2 universes of data will be received) is easy but for example in Test 2 knowing which universe will be received first from the 2 senders is more difficult. There wasn't time to create and test a more complicated mechanism for doing the checks automatically so therefore these tests are checked manually based on the "expected-results.pdf" table. This keeps the leg work down to a minimum in that the user must just check the file outputs and doesn't have to worry about how to run the tests.

11.2.3 Fuzz Testing

The integration and unit tests are both focused on checking behaviour of the protocol in specific conditions. What this doesn't check however is the behaviour of the protocol when given a wider variety of inputs. Where this is particularly important is with the packets parsed from the network. It is possible that there may be multiple different protocols operating on a network and so therefore it is possible that the implementation may receive packets from these sources. It is also possible that there may be malfunctioning/malicious sources on the network sending random or scrambled data. This means that the protocol should be able to handle this by flagging up

malformed packets without crashing. This is particularly important in this case because this library may be used within an implementation of a show-critical device and so therefore should avoid crashing as much as possible. To test how well the library does in this regard a technique called fuzzing is used. This is where a fuzzing program generates data (based on some initial inputs) which it then feeds to the program being tested and it checks if the program crashes. This fuzzing program does this continuously using a huge variety of possible data while recording how the program handles it each time.

For this test the american-fuzzy-lop library [41] was used. This was setup as described at [42] and run on the Fedora 31 operating system. The fuzz target code used is included in the sacn-parse-fuzz-target subfolder of the Fuzzing folder. This code is extremely basic and just passes the provided fuzzing data straight to the parse function and ignores the result. This therefore doesn't check the error returned or if a specific packet is parsed but it doesn't check that the library parsing mechanism runs without encountering a crash (a rust panic!).

To guide the fuzzer to produce data based on sACN expected packets the raw data for a data, synchronisation and discovery packet are used as inputs. These packets are found in the 'fuzz_in' subfolder of the Fuzzing folder. The packets were generated by performing a wireshark capture of the implementation sender sending these packets and this is included as the "fuzz-test-base-captured-packets.pcapng" file. These captured packets were then transformed into raw data files utilising the wireshark export raw data export feature as described in [43].

The fuzzer was run with 2 separate instances with run conditions and the results of the fuzzer shown in figure 18. These results show that out of the 218 million packets tried 98.7 thousand produced a crash with 14 unique 'types' of packet generation that caused a crash. This corresponds to a calculated crash rate of 0.045% based on the inputs generated by the fuzzer. While ideally the project would have a crash rate of 0% this crash rate is still low enough to be acceptable.

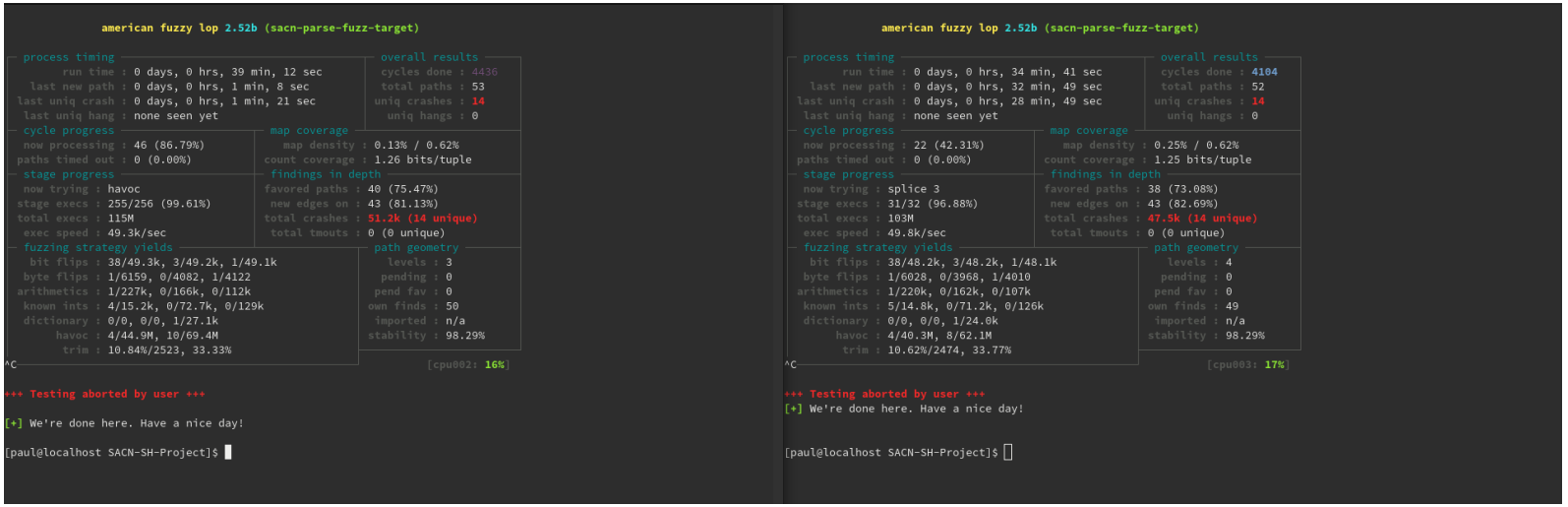


Figure 18: The results of the fuzz testing on the parsing part of the library.

11.2.4 Testing External Interoperability

The unit and integration tests show that the program works within itself but it is unlikely that within a deployment scenario only a single implementation would be used and so therefore it is also required to show that the library is interoperable with other programs. Since all the programs run the same protocol it is expected that they should all be able to communicate.

These tests can highlight problems with the program which are hidden until this point such as gaps in the specification where the library behaviour is implementation defined and may not be compatible with other systems. It can also highlight parts of the system which perform slightly differently than described in the abstract specification due to the introduction of real-world factors such as real-equipment limitations like processing speeds. An example of this might be if the library absolutely relied on universe discovery packets being sent at exactly the interval as defined by the specification. In-real systems network delays as well as varying workloads on the devices might cause packets to be received at slightly variable intervals. Real-world tests therefore help find some of these problems and allow fixes to be made before the program is sent to users.

Industry Sender - Avolites Titan Setup To allow repeatability the show files used for the interoperability tests are included in the Avolites Titan Show Files sub folder in the Interoperability Testing folder. These show files were made for version 11.4 and run on an Avolites Titan Mobile. Within the show file each universe was assigned to an sACN universe with a 1 to 1 mapping and no other network protocols were used as shown in figure 19. In addition to this within the show-file itself 1 channel lighting fixtures called 'dimmers' were used to allow a 1:1 mapping between a fixture in the show-file and a DMX-address. This mapping is shown in figure 20 which shows some of the dimmers used (the groups part of the window shows that 8190 dimmers were added to represent each channel in universes 1 - 16). All settings used are included within the save files but in general were left to their defaults.

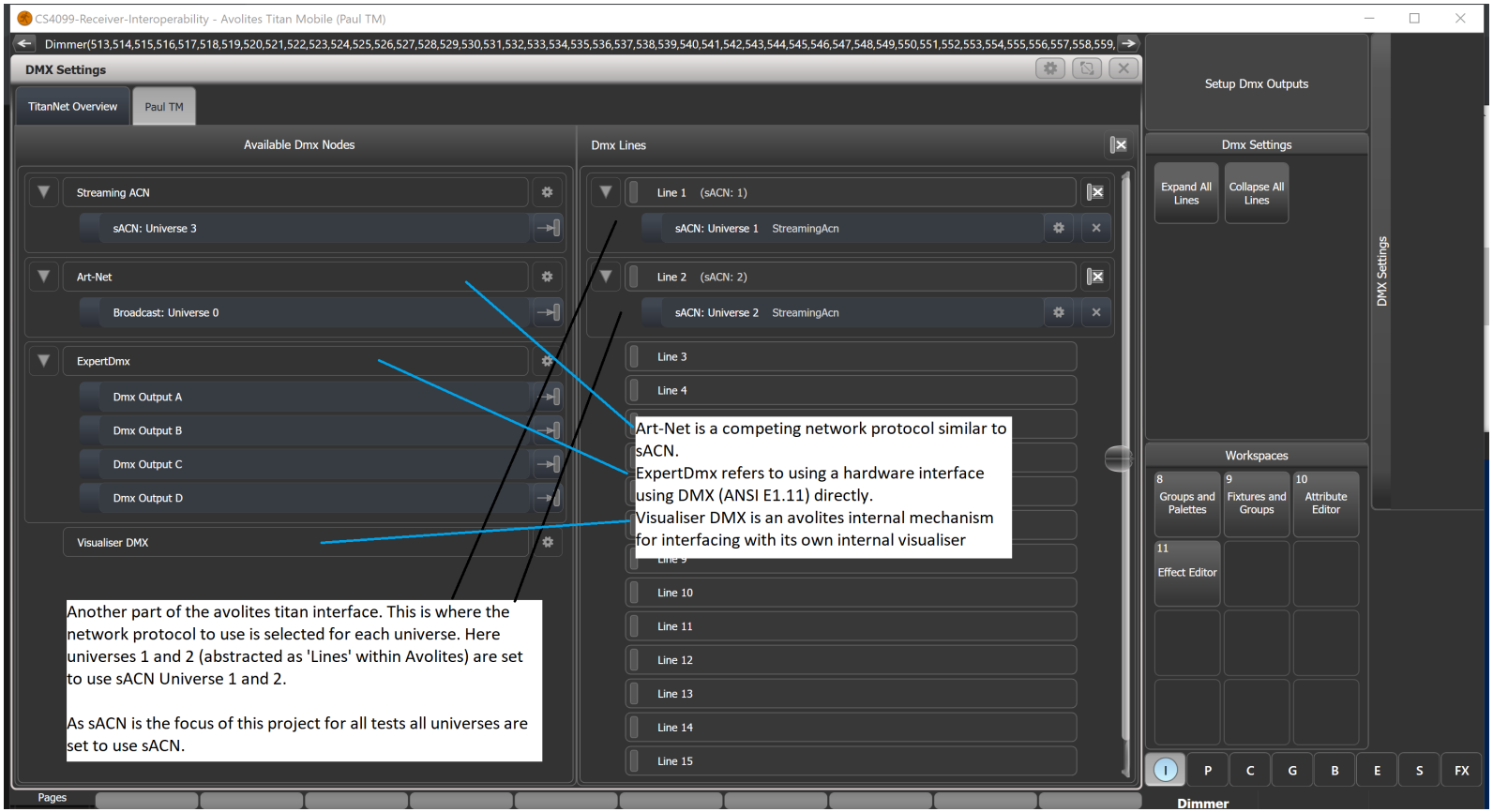


Figure 19: The setup of the avolites show file network protocols used for the interoperability testing

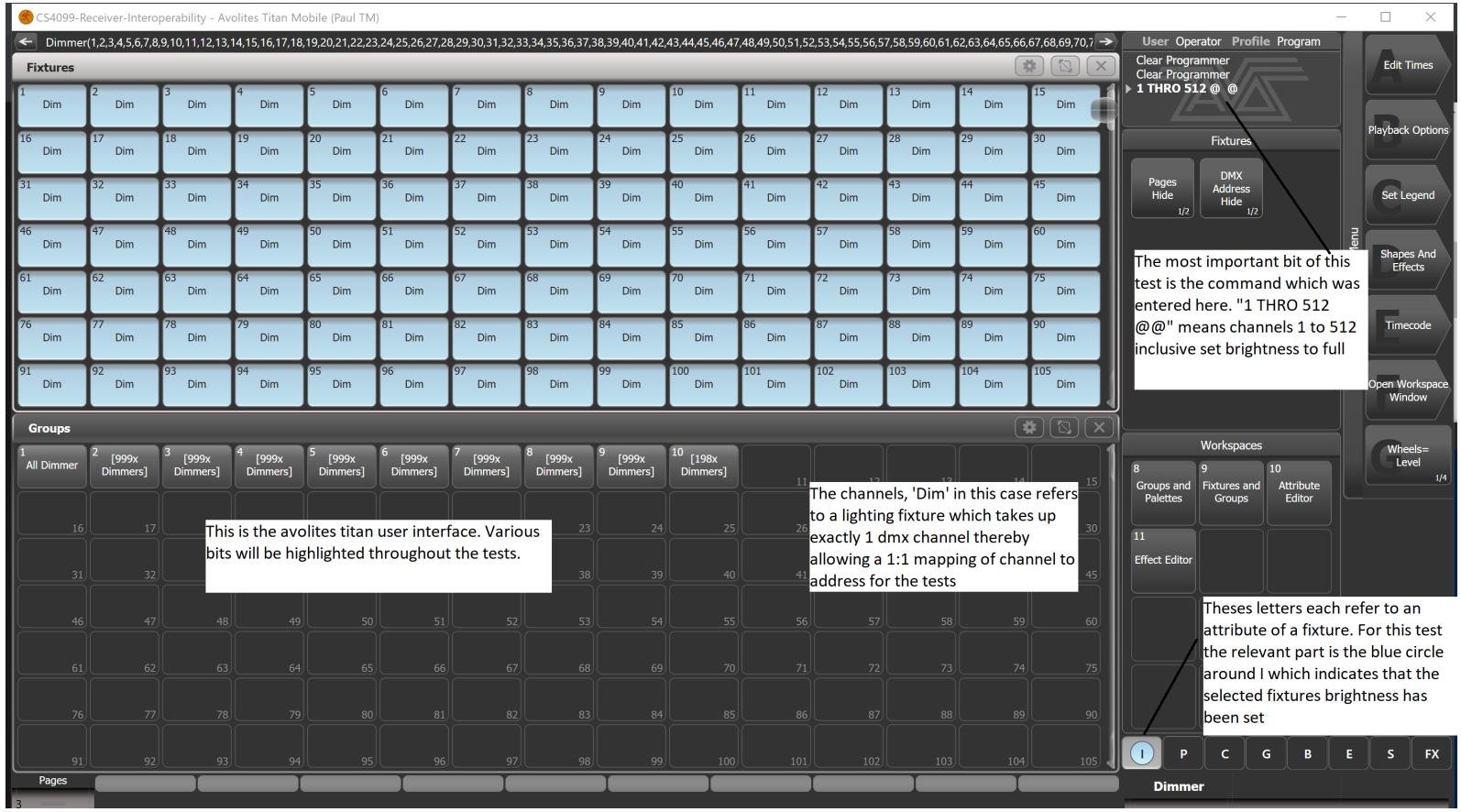


Figure 20: The setup of the avolites show file used for interoperability testing showing some of the fixtures patched

Industry Receiver - Vectorworks Vision Setup The vectorworks vision visualiser uses the vision files "CS4099-TEST.v3s" and "Student-Union-Model.v3s" included within the Test Resources folders of the Sender Interoperability Testing and Acceptance Test folders. Vision was setup using medium graphical quality settings although this should have had no effect on the results. The patch used within each file is included within the file itself as is the positions of each individual fixture. The 'DMX Provider' setting was set to sACN for all tests. The tests were performed on Vectorworks Vision Plus 2019 with a professional license and version 24.0.6.521266.

Industry Receiver - sACNView Setup sACNView was setup using the default settings with the ethernet interface assigned to 192.168.0.6 set as the network interface as shown in figure 21. Universes 1 - 16 inclusive were listened to for every test even if less than that were used for a specific test. Unicast and multicast were enabled for every universe as shown in figure 22.

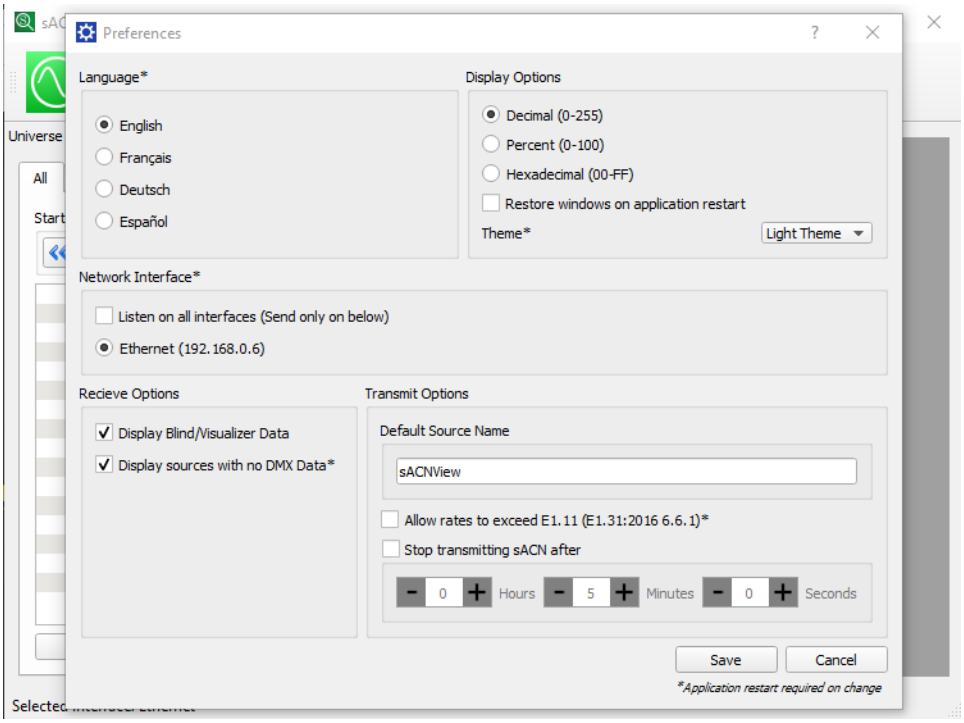


Figure 21: The setup of sacnView used for the interoperability tests

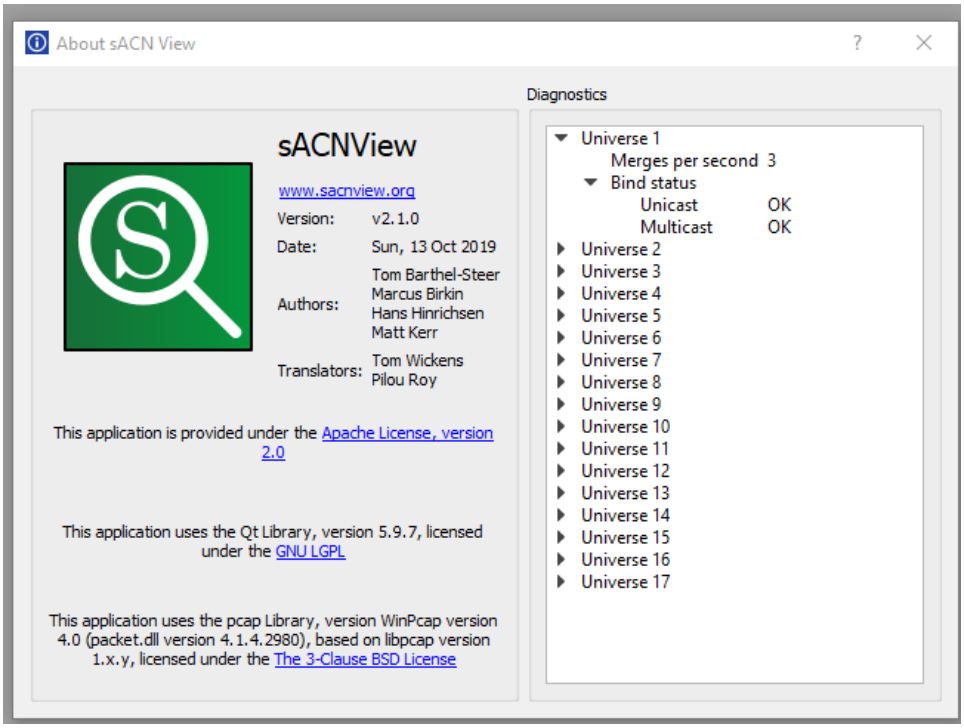


Figure 22: The IP modes enabled in sacnView used for the interoperability tests

Testing receiver implementation To show that the sACN receiver implementation is interoperable with a real-world sender the demo receiver program was set-up and run in a network as shown in figure 23 along with an professional industry source of sACN in the form of the Avolites Titan program as described in the tools section. The tests run and what they demonstrate are detailed in the included "CS4099 - Interoperability Testing.pdf" document. The real-world implementation used doesn't support sending universe synchronisation or universe discovery data so these could not be tested in this step however details of how theses would have been tested are also included in the document.

For some of the tests it was easier to determine if they passed by visualising the received data. In test 5 the sender sends data on 16 different universes with different value ranges per universe. For this test to pass each universe should only contain the values within the range assigned to it as specified in the testing document. To show that this was the case the results were output to a csv file "test-5-out.csv" and then processed using a spreadsheet "test-5-data-processed.xlsx" to produce the graph "test-5-processed-first-value-chart.png" which is included in figure 24. This graph shows that each universe stays within the range expected and therefore that the test passes. As a sanity check of the results this test was also repeated using the

sACN-viewer as the receiver and the graph produced in real-time recorded and included as the "Test-5-Receiver-Control-sACN-Viewer.mkv" file.

Similar to test 5, tests 3 and 4 also included visualisation elements as part of the check and details of these as-well as further specifics/details of all tests are included within the screenshots/videos within the relevant folders test folders (excluded from report for brevity).

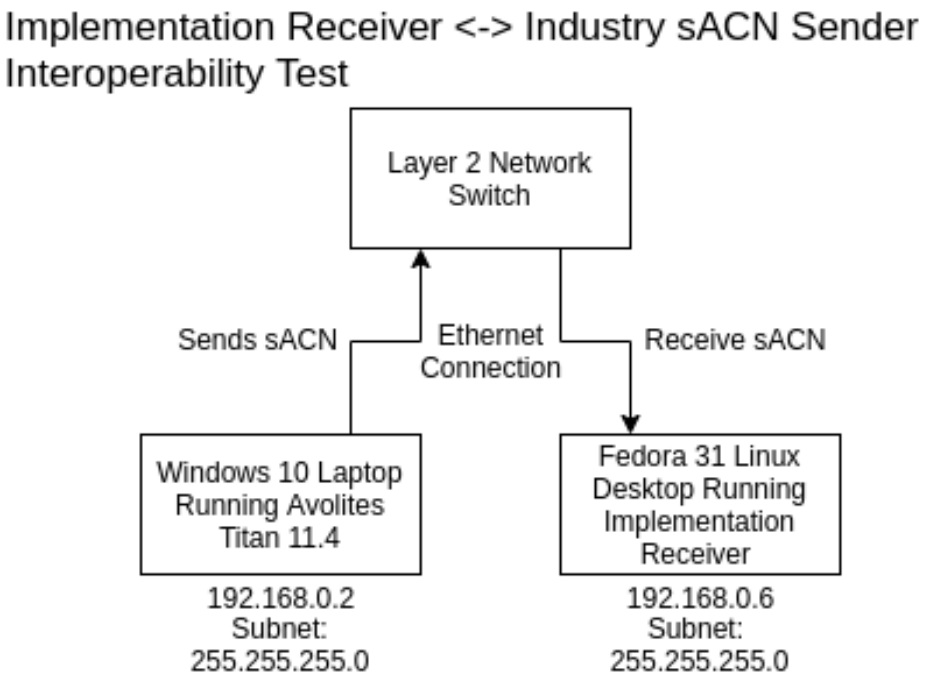
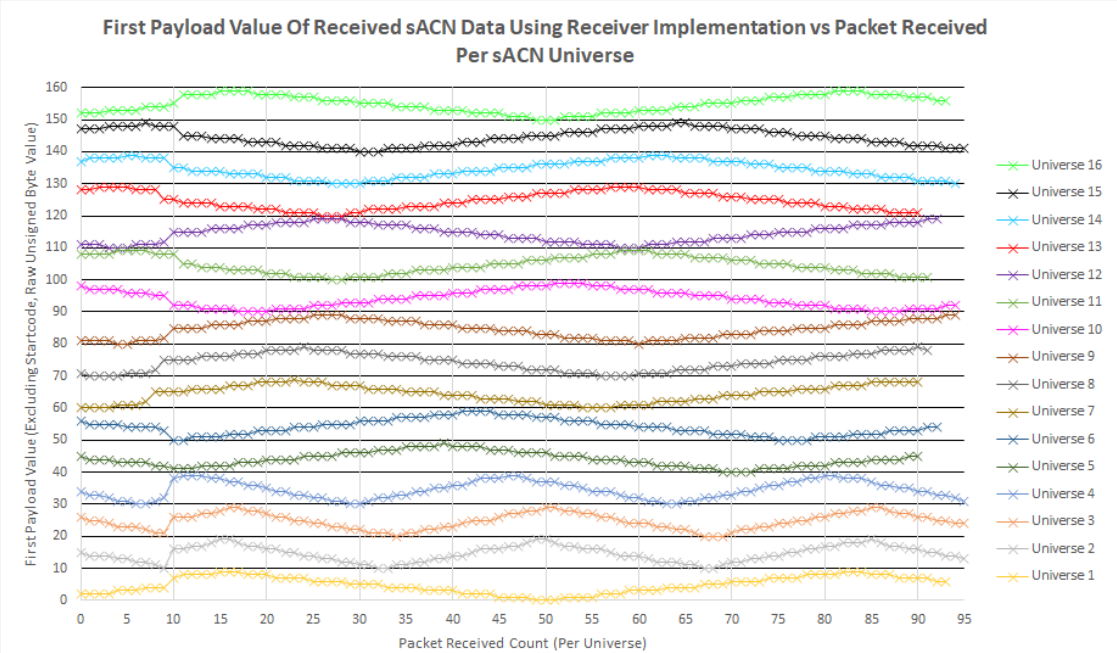


Figure 23: The set-up of the text with the implementation receiver and an industry sACN source



Values for each universe are shown to vary within the expected ranges and not overlap. This therefore means that the receiver is receiving each universe correctly.

This graph can be compared to the real-time graphing produced by sACNView shown in the included video to see that it looks approximately similar.

Note that the sACN viewer output and the receiver output were not recorded at the same time and so therefore the exact position of points compared between the two is meaningless. The important part is that the values for each universe are staying within the ranges expected indicating the receiver is not mixing up data from different universes.

Figure 24: A graph showing the values of each universe per packet received for that universe. This shows the ranges of each universe fit within the expected.

Testing sender implementation Similarly to test the sACN sender implementation an external receiver was used. In this case 2 separate receivers were used, the vision visualiser and sACN viewer, as discussed previously. Separate programs were used because neither program was suitable on its own. The visualiser was created by a large company within industry and is used every day by professionals working in the field. This gives a high confidence that it will be compliant with the protocol and so showing interoperability with this is very valuable. It was found however that the visualiser only supports data packets and does not support universe synchronisation or discovery (at least in a way that could be observed) meaning it could not test this functionality. sACN viewer was therefore used as it provides support for universe discovery as-well as a good interface to show that data packets are being received and parsed correctly. Unfortunately neither implementation supported universe synchronisation. Given that the same problem was encountered when trying to find a receiver implementation it appears that the industry has not yet fully caught up to the ANSI E1.31-2016 standard when synchronisation was added. It may also be possible that while in-use programs supporting this feature do exist they are proprietary and so could not be accessed/tested against within this project.

The test was set-up identically for both receivers as shown in figure 25, the actual tests run are detailed in the "CS4099 - Interoperability Testing.pdf" document along with results. This figure also describes the tests which would have been run had there been an industry receiver which supported the universe synchronisation feature.

To allow the visualiser output to be easier to interpret the 3D scene used uses a large number of simple lighting fixtures. These fixtures are laid out as shown in figure 26 with each fixture corresponding to a single channel on a universe.

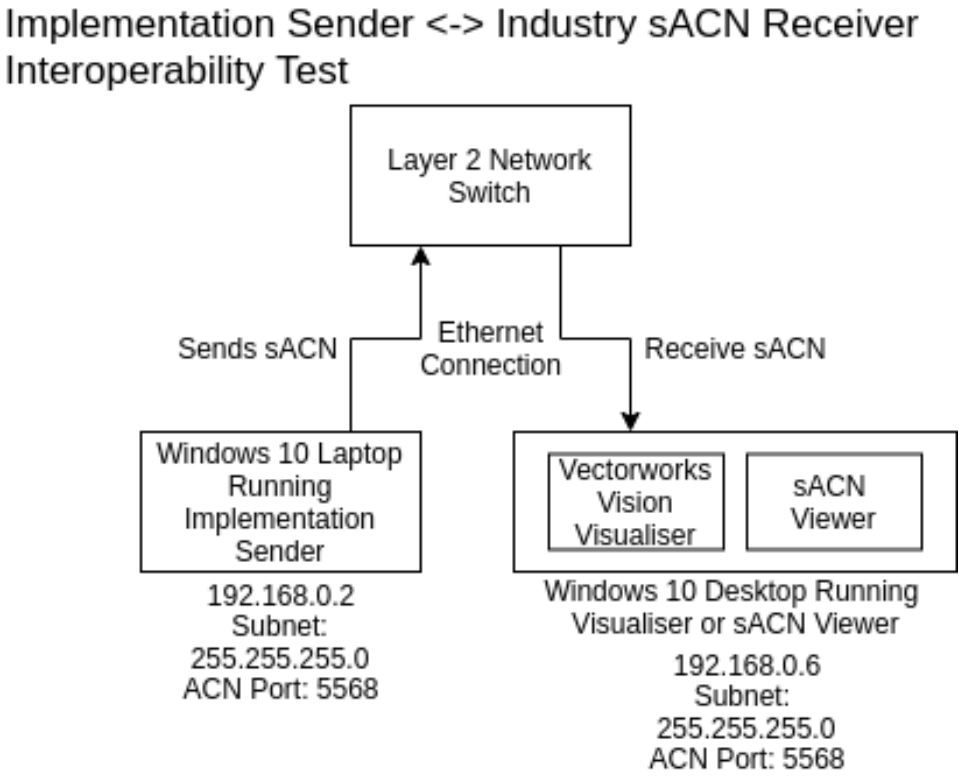


Figure 25: The set-up of the sender implementation interoperability test

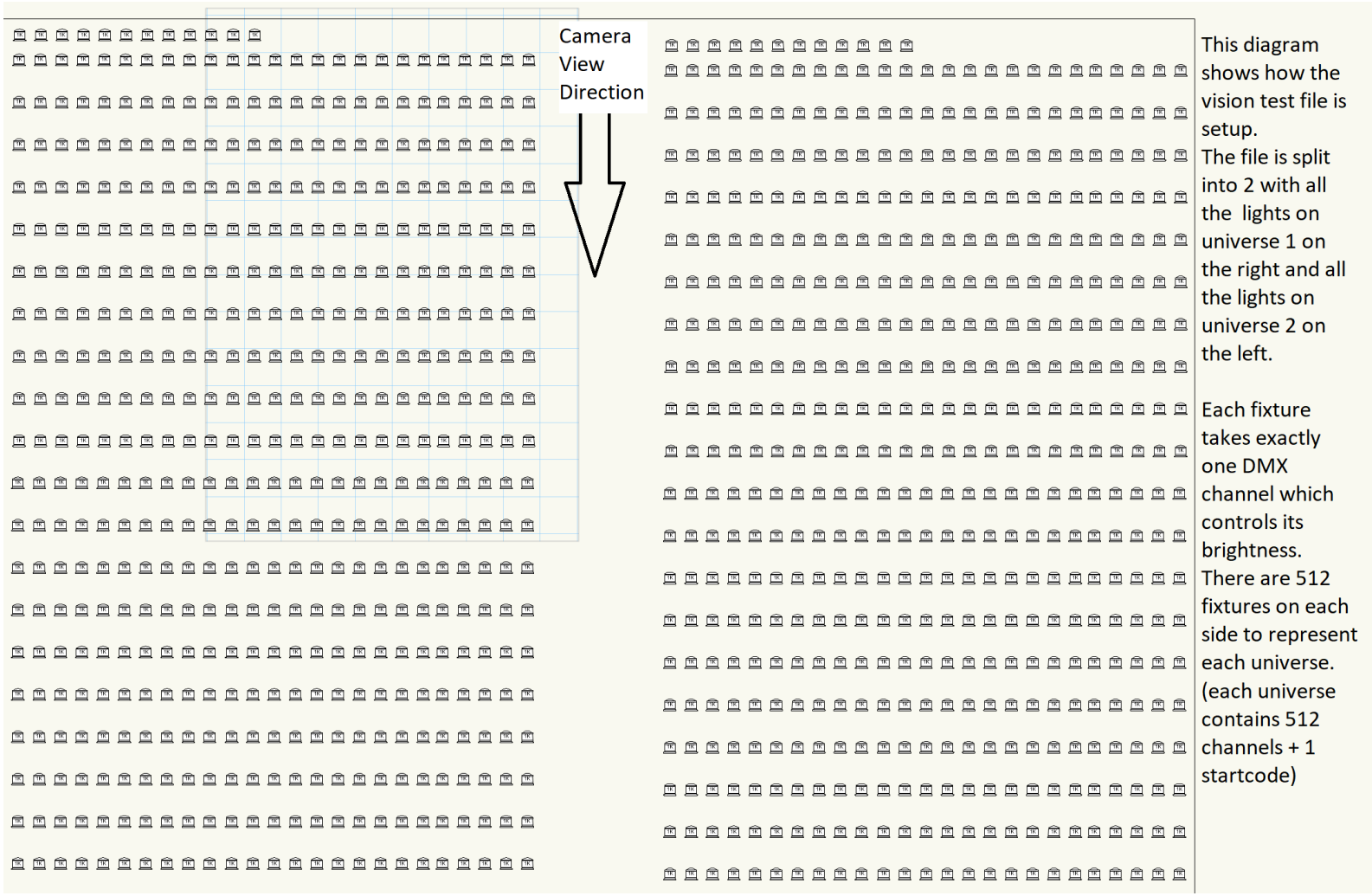


Figure 26: The lighting plot of the lights used as part of the Vision Visualiser test scene

11.2.5 Acceptance Testing

Acceptance testing is the final stage of testing within the project and represents the transition from the testing phase to the deployment phase. For this test a similar setup is used to the external interoperability tests however this time rather than just the developer these tests were performed in the presence of an industry professional. This allows demonstrating that the program can actually be used for its intended purpose, it also allows the chance for people in industry (the targeted end users) to provide feedback or evaluation about the usefulness of the program and point out potential problems.

Two separate demonstrations are performed, first showing the functionality of the program as a sender with data sent from the implementation to the visualiser with the person seeing both the commands entered into the implementation sender and the results on the visualiser. The second demonstration shows the sACN source lighting board (Avolites Titan) sending to the receiver with the data received displayed on the screen in text format.

The industry professional for this test was the Technical Supervisor for the St Andrews Students Association. As a technician they work with lighting, sound and other entertainment systems daily and so they are ideally placed to demonstrate the implementation of this widely used lighting protocol to. The visualisation test provides significant value to the project as a professional working in the field will know how this fits into the real-world work flow of someone working in lighting and therefore that this is an actual representative usage of the protocol. The receiver output from the lighting board is also extremely valuable as the technician is able to observe that this is a real-world sACN source which is being used with the protocol and they can see that the data is being correctly sent by the board and received as expected.

The test layout is shown in 27, the plan for the demonstrations to run are detailed in the "CS4099 - Interoperability Testing" however during the actual test there is the possibility of questions which may lead the demonstration to change to show specific areas of the implementation. Once this test was complete the professional then agreed to write up a short email describing the test and their evaluation of the demonstration.

The lighting layout used for visualisation in this test is setup to be similar to the actual setup used in the students union thereby being a representative example of an actual industry use case. The layout of this setup with accompanying explanation is shown in figure 28.

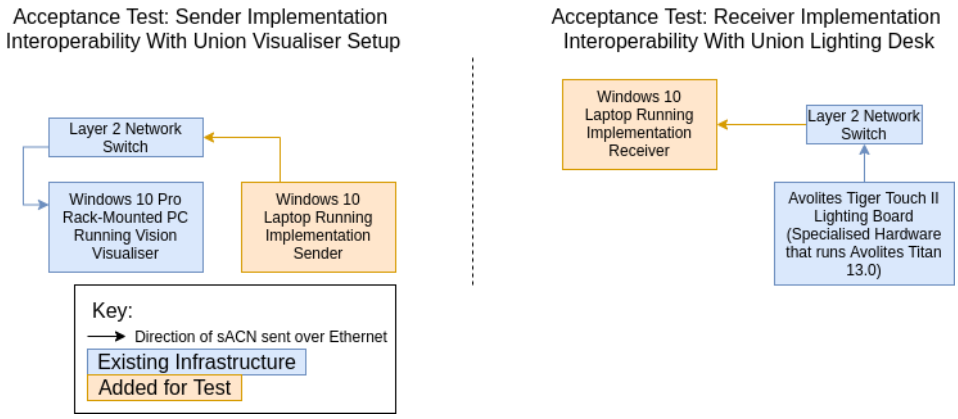


Figure 27: The layout of the acceptance test performed

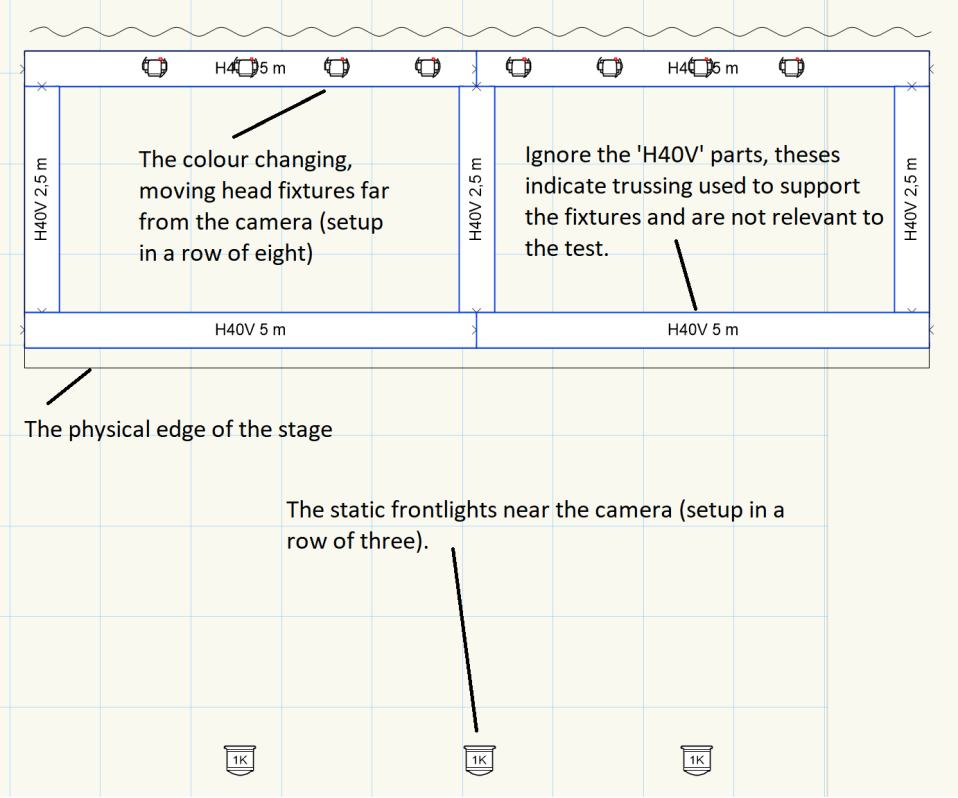


Figure 28: The lighting plot showing the locations and addresses of fixtures within the visualiser used as part of the acceptance test. This is based on the actual lighting layout used within the students association. Additional explanation has been added on top of the plan to describe what each part means.

11.3 What Testing Shows

The unit and integration tests in combination with the code-coverage show that the code works as expected but to show that the behaviour actually fits with the protocol specification compliance testing was performed. Ideally this would be done through an external compliance test suite however none exist public-ally for the protocol. Therefore a compliance suite was created, this was done by going through the protocol specification document [3] and generating a list of required functionality for each section, unit and integration tests were then created so that each requirement was fulfilled. This systematic approach makes it much less likely that something will be missed and increases confidence that the implementation will be compliant with all aspects of the protocol. This table is included in the attached "ANSI E1.31-2018 Compliance Check List.pdf" document coloured coded to show the results of the tests for each requirement. Further to this the interoperability tests also provide evidence that the implementation is compliant because the programs used have been shown to be compliant themselves so being interoperable through the protocol indicates the implementation is compliant. The acceptance test then provides evidence that the implementation can actually be used as expected. This testing therefore shows the progress from design to implementation through to a deployable compliant implementation.

12 Evaluation and Critical Appraisal

sACN Project Requirement Checklist			
Requirements	Level of support		
FUNCTIONAL	Attempted	Implemented	Tested
Sending start-code/DMX data over sACN			
Receiving start-code/DMX data over sACN			
Sending synchronised DMX data			
Receiving sychronised DMX data			
Sending universe discovery adverts			
Receiving universe discovery packets			
NON-FUNCTIONAL			
Demonstrated deployment of the library in a real-world system			B
Windows 10 Support		A	
Fedora Linux Support			
Unicast Support			
Multicast Support			
Broadcast Support			
IPv4 Support			
IPv6 Support			

Key:
Green indicates Yes / tick
Orange indicates functionality has limitations
Red indicates not met

A: Windows receiver does not support IPv6 Multicast Receiving with sACN, all other listed functionality supported
B: As discussed in COVID-19 section this was done to the best of my ability given the situation.

Figure 29: A table showing the features attempted, implemented and tested as part of this project based on the requirements

Figure 29 shows the support for the library with respect to the requirements specified at the start of the report. This tables shows that almost all of the requirements have been up-to the point of being implemented and then verified through testing. The table shows that 1 of the non-functional requirements was not fully implemented and this was due to a lack of support provided by the rust library used for IP communication. The requirements specified at the start of this project differ slightly from those presented in the DOER at the start of the project as shown in figure 30. This came as part of a supervisor change half-way through the project which meant that

the requirements were re-evaluated. The primary objectives were condensed from 5 points to 3. This didn't actually represent a specific change in requirements but more so that requirements such as 'Learn rust' and 2. were implicit parts of the project so didn't need to have their own requirement. Within the secondary objectives 1. and 2. were removed. The reason for this is that it was deemed that at the half-way point that these requirements were not going to be realistically reachable to a high standard as they could form the basis of entire projects in and of themselves. The removal of these requirements didn't effect the software engineering process, the reason for this is that no work had been started on these requirements as they were both part of the 'testing' phase and so they could be easily dropped. Overall the project meet the majority of its requirements as specified at the start and all of the later refined requirements and so therefore the project can be considered a success.

Objectives

Primary Objectives

1. Learn rust

2. Establish the extends of the existing implementation and introduce more thorough testing to verify its correctness (fixing issues found).

3. Expand the existing implementation to allow receiving DMX through sACN.

4. Expand the existing implementation to allow sending DMX with universe synchronisation.

5. Expand the existing implementation to allow universe discovery.

Secondary Objectives (extensions on top of primary objectives)

1. Analyse the performance of this protocol implementation verses other implementations in different languages using a number of different metrics.

2. Analyse the performance of this protocol in various implementations verses another protocol such as ArtNet [7].

3. Test the completed implementation in a real-world environment by creating a demo program which utilises the library.

4. Window and Unix support.

5. Introduce unicast and broadcast support.

6. Ipv4 and Ipv6 support

Figure 30: The requirements of the project as listed in the DOER at the start of the project

There exists no fully-implemented public-ally available implementation of sACN in rust and the most complete version was used as the base of this project, this means that there is no direct comparison possible between this project and another however there do exist implementations of sACN

in other languages so these can be used for comparison. Of the libraries found many do not provide support for universe synchronisation as they are based on the older ANSI E1.31-2009 from before synchronisation was added. Many of the libraries also take a more bare-bones approach from the perspective of the user by exposing the packet structure directly. This requires more learning by the user about how sACN works especially as it means all the synchronisation and discovery behaviour isn't implemented and must be handled by the user. The lack of synchronisation support potentially explains the issues found during interoperability testing when it came to finding implementations to test against that did support synchronisation. The most complete public-ally available library seems to be the open lighting project (C++) [10], which provides support for transmitting and receiving sACN and provides support for Linux, Mac OS and FreeBSD meaning it is tested for more operating systems than this rust project. The library does not support windows however which this project does (except for the IPv6 windows receiver limitation). The feature comparison between this project and other libraries is shown in figure 31. This figure shows that this project is more complete than many of the implementations. This is unless support is required for a particular system such as arduino/bsd or the system must be very lightweight e.g. the C++ impl. The lack of a fully complete implementation in any language is likely to do less with one existing and more than the companies that created it probably didn't release the code as they gain an advantage by being able to offer features that others can't. The choice for this project to only focus on Windows and Linux (specifically Fedora 30/31) was based on a lack of test devices to use in any other system.

Feature	This Project	C/C++ Impl [1]	Arduino Impl [2	Node Impl [3]	Python Impl [4]	Open Lighting Impl [5]
Data sending, no synchronisation						
Data receiving, no synchronisation						
Synchronised data sending					C	
Synchronised data receiving (even if just parsed and passed upto user to handle)						
Synchronised data receiving (including fully synchronisation handling behaviour)						
Termination Packet Sending						
Termination Packet Receiving						
Termination Packet Behaviour Handling (beyond just parsing a data packet with the termination option as normal)						
Discovery packet sending						
Discovery packet sending (with automatic sending at the correct ANSI E1.31-2018 interval)						
Discovery packet receiving (with discovery list rebuilding from multiple pages)						
IPv4 Support						
IPv6 Support	A					
OS Support - Windows					B	
OS Support - Linux						
OS Support - Other			Arduino			FreeBSD, Mac OS
IP Unicast						
IP Multicast	A				B	
IP Broadcast						

Key:
Green indicates feature present, orange indicates feature some-what present, red indicates feature missing or not marked as supported and tested, grey indicates unclear from documentation.

Feature lists as of 21/04/2020.

A: Cannot do IPv6 multicast receiving on windows (other IP modes still work)
B: Described as 'a bit tricky'
C: Not fully compliant with the protocol recommendations as specified in the documentation "this is not implemented like the recommended way"

[1] https://github.com/hhromic/libe131
[2] https://github.com/forkineye/E131
[3] https://github.com/hhromic/e131-node
[4] https://github.com/Hundemeier/sacn
[5] openlighting.org/ola/

Figure 31: A table showing a comparison between this project and other similar projects in terms of sACN features implemented

The decision not to pass up data packets awaiting synchronisation means that the packets must be temporarily stored within the receiver and this is done using a Vec data-structure which is a dynamically sized structure. This means that the memory allocated to the program will continue to increase as packets are received which can be problematic for embedded devices with limited memory capacity. To limit this problem the implementation relies on the limited number of possible universes in the protocol and only stores a single universe of data for each waiting universe. This limits the maximum required space for this storage to 31.3MB + overhead which is not a problem for any modern PC but is potentially a significant amount for an embedded device. This means that the library is at risk of running out of memory for some devices such as arduino [26] which are commonly used for creating simple DIY embedded systems. This is only a risk on systems which have a large number of universes being synchronised at once so the for majority of usage cases where most universes aren't synchronised and only

a few are synchronised at any one time this isn't a problem. To avoid this problem on an embedded system it is therefore required to keep the number of universes being listened to 'low' (other universe packets are discarded) with 'low' decided by the resources available (based on benchmarks etc.). This potentially explains why the arduino implementation of sACN [7] does not support universe synchronisation.

$$max\_possible\_universes \times universe\_capacity = 63999 \times 513B \approxeq 31.3MB$$

The library is based on the std-environment. This means the implementations utilises the standard rust crates to provide functionality such as hash-maps and threads. This decisions means that the produced binary is potentially bigger than it otherwise might be and isn't as tuned to the specific application from the perspective of performance. These costs come at an advantage however as re-using standard libraries means that the required features don't have to be re-implemented from scratch. This limits the testing required and reduces the chance of bugs as the existing implementations are already widely used and tested. It also reduces the development time required significantly which was vital to allow this project to be completed within the time-allowed. It would not have been possible to create the library within the given-time without utilising at least some existing libraries/implementations such as std.

The receiver uses a single threaded design with the timeout for all source + universe sequence numbers being checked when any sequence number is checked. As every source/uni combination is checked every-time a sequence number is checked this comes with a performance hit as they all must be visited each time. This is required because otherwise a source which has completely stopped transmitting on a universe and for which the termination packets are lost would never be removed from the sequence numbers and would take up space on the receiver continuously which is problematic for embedded devices. While not used within this implementation an alternative strategy could be to only check time-outs occasionally (say every 5 sequence number checks) or to have the timeout checks be done periodically based on a time interval. This would reduce the number of checks required and therefore theoretically increase performance at the cost of having dead-universe and source sequence numbers stored longer than is required.

The fuzzer test outcomes highlight an area of potential improvement, this would involve using the generated packets which caused the crashes to track the problematic code and introduce fixes to lead to a more robust and fault-tolerant parsing system. There wasn't sufficient time to do this in any significant depth within this project however this is a potential area for further work.

Couldn't show that the receiver worked with universe synchronisation or discovery. This is because the real-world sACN sender used (Avolites Titan) does not support it. Ideally another source would have been used however an initial inspection found none that did support it that could be used. If there had been time a test program could have been written using a library in another programming language and this used however there was insufficient time to learn, write and test an entirely new library in another language so that it could be used for this test.

13 Conclusions

An implementation was successfully created with supports the ANSI E1.31-2018 sACN protocol in the Rust programming language including the newer (2016 onwards) features of universe synchronisation and universe discovery. This implementation was extensively tested from the perspective of both correctness within itself and for compliance and interoperability with the protocol and other compliant devices. The implementation provides support for most of the non-functional requirements specified at the start by supporting IPv4, IPv6, Unicast, Multicast, Broadcast, Windows and Linux. The biggest drawback to the implementation as it stands is the lack of IPv6 multicast support on the receiver side in Windows.

For future work there are two directions to propose taking the project in. The first is from the perspective of continuing to increase the library support itself, this could take the form of supporting more devices but I believe the next step to look at is integrating the features of ANSI E1.33 (RDMnet) into the library. This is because driven by support from major companies such as ETC the features provided by E1.33 are likely to be become a major part of the lighting over IP eco-system in the next few years. Rust does not have any implementation of E1.33 public-ally available so if extended the library would be the only library with support. The other direction is in using the library to make devices/software for usage with sACN. Rust provides a high level of performance while maintaining many safety guarantees which would make it an ideal language to create show critical high performance lighting control software that is robust and usable on multiple platforms. By creating this library it opens the door to the development of this software in the language and the potential benefits this brings.

14 Appendices

After the references more appendices are appended to show more details of the project. These documents are referred to as relevant throughout the report and attached so that they are all within one report file. The documents are also included as separate files within the project resources submission.

14.1 Extraneous Circumstances - COVID-19

Unfortunately the acceptance test which was planned for sometime between the 14th and 29th of March based on when the technician was free was unable to take place. This is due to the outbreak of the COVID-19 virus which forced the students association to close and non-essential contact to be stopped before the demonstration could be conducted. It is predicted that based on the very similar integration tests passing as-well as the other testing that had the demonstration gone ahead the program would have worked as expected and the evaluation been positive. It is a shame that the test could not be conducted as comments could have led to recommended improvements for the project which would have ultimately resulted in a higher quality submission. An annotated video is included showing what one of the acceptance tests may have looked like is included although this video is performed on my own hardware rather than the students union equipment.

The COVID-19 also meant that I was unable to remain in St Andrews and continue to have access to the computing labs. This had a number of impacts. The first was the time lost due to the move and having to create a new setup for working at home. This environment was less than ideal and meant that some features which may have been possible had things continued as normal were not. The move also meant that I no longer had direct access to more than 1-2 machines. This means that tests like the ssh integration tests were much more difficult to create because they had to be done completely remotely from home SSH'd into labs. This lead to far fewer of these tests being created than would have been hoped.

14.2 User Manual

The details of how to run the various tests, demo-programs and examples are described in the "usage.pdf" file. Installation instructions are detailed within the README file.

The core part of this project was the sACN library created. This library is packaged as a rust cargo crate and therefore can be imported using a local

import as demonstrated in the 'demo_src' and 'demo_rcv' programs. After this project is complete the project will hopefully be uploaded to the public rust cargo repository which would allow much easier installation through the cargo tool-chain and fetch mechanism.

Usage of the library is described in the generated rust-doc documentation. Once the project is complete this would also be bundled with the library in the public cargo repo to allow easier access however as that cannot be done until after this project is marked the documentation is included in the sacn subfolder of the Code Documentation folder. Within this folder the documentation can be opened as a web-page by opening the index.html file. This documentation contains details of the functionality of public and private functions as-well as the possible returned errors, examples of the code in usage and also includes the 'demo_src' and 'demo_rcv' crates. As this documentation includes private code it the web-pages are bigger than they would normally be but it was decided that for the purposes of submission the full documentation was more suitable. In actual usage an external user of the library would normally just compile the documentation using the same command (as detailed within usage.pdf) but without the "–document-private-items" argument so that only the public documentation is generated. The private documentation is then for those who are developing the library itself.

To check that the documentation contains the right information it was compared to the documentation for another sACN library [6] and this is summarised in the table in figure 32.

Documentation Area	C++ Implemention	This Project
General description of the library.	Yes	Yes, included in cargo documentation
General description of sACN.		Yes, At start of report, this will be moved into a seperate document for inclusion in library after submission and marking is finished.
Installation Instructions.		Yes, in the README document.
Details of the various errors		Yes, included in cargo documentation
How to Unicast Transmission		
How to Multicast Transmission		Yes, Included as an example in cargo docs
Public constants documentation		Yes, included in cargo documentation
Public functions documentaiton		Yes, included in cargo documentation
Example to create sACN receiver (client)		
Example to create sACN sender (server)		Yes, Included as an example in cargo docs
License		Yes, At top of files

Figure 32: A table showing a comparison between the documentation of another sACN library [6] and this project, green indicates that the area is covered

References

[1] ANSI E1.17 - 2015 Entertainment Technology - Architecture for Control Networks, CP/2011-1007, 21 May 2015.

[2] User: lschmierer. (2018, April. 20). Streaming ACN implementation for Rust [Online, Accessed: September 2019]. Available: https://github.com/lschmierer/sacn

[3] ANSI E1.31 - 2018 Entertainment Technology Lightweight streaming protocol for transport of DMX512 using ACN, CP/2014-1009r6a, 7 November 2018.

[4] ANSI E1.11 - 2008 (R2018) Entertainment Technology - USITT DMX512-A Asynchronous Serial Digital Data Transmission Standard for Controlling Lighting Equipment and Accessories, CP/2007-1013r3.1, 31 May 2018.

[5] User: shabaz. (2017, Aug. 24). DMX Explained; DMX512 and RS-485 Protocol Detail for Lighting Applications [Online, Accessed: 17/09/2019]. Available: https://www.element14.com/community/groups/open-source-hardware/blog/2017/08/24/dmx-explained-dmx512-and-rs-485-protocol-detail-for-lighting-applications

[6] https://github.com/hhromic/libe131 (17/09/2019)

[7] https://github.com/forkineye/E131 (21/04/2020)

[8] https://github.com/hhromic/e131-node (21/04/2020)

[9] https://github.com/Hundemeier/sacn (21/04/2020)

[10] https://www.openlighting.org/ola/ (21/04/2020)

[11] https://www.rust-lang.org/ (17/09/2019)

[12] http://artisticlicence.com/WebSiteMaster/User%20Guides/art-net.pdf (17/09/2019)

[13] https://docs.rs/sacn/0.4.4/sacn/index.html (26/01/2020)

[14] https://tsp.esta.org/tsp/documents/docs/E1-31_2009.pdf (26/01/2020)

[15] https://tsp.esta.org/tsp/documents/docs/E1-31-2016.pdf (26/01/2020)

[16] http://www.rdmprotocol.org/files/What_Comes_After_Streaming_DMX_over_ACN_%20%284%2 (26/01/2020)

[17] RDM-NET

[18] https://github.com/ETCLabs/RDMnet (26/01/2020)

[19] RDM

[20] https://www.etcconnect.com/About/ (26/01/2020)

[21] https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust.html (28/01/2020)

[22] https://www.techrepublic.com/article/rust-programming-language-seven-reasons-why-you-should-learn-it-in-2019/

[23] https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm (01/01/2020)

[24] https://tools.ietf.org/html/rfc894 (10/03/2020)

[25] https://doc.rust-lang.org/1.7.0/book/no-stdlib.html (11/03/2020)

[26] https://www.arduino.cc/en/tutorial/memory (11/03/2020)

[27] https://doc.rust-lang.org/std/macro.try.html (12/03/2020)

[28] https://github.com/rust-lang-nursery/error-chain (12/03/2020)

[29] https://www.wireshark.org/ (12/03/2020)

[30] https://sacnview.org/ (12/03/2020)

[31] https://sacnview.org/documentation.html (12/03/2020)

[32] https://www.vectorworks.net/en-GB/vision (12/03/2020)

[33] https://www.avolites.com/product/titan-mobile/ (12/03/2020)

[34] https://www.dcs.bbk.ac.uk/ ptw/teaching/IWT/transport-layer/notes.html (13/03/2020)

[35] https://tools.ietf.org/html/rfc5771

[36] https://tools.ietf.org/html/rfc2365

[37] https://tools.ietf.org/html/rfc4291

[38] http://softwaretestingfundamentals.com/unit-testing/ (06/04/2020)

[39] https://github.com/mozilla/grcov (06/04/2020)

[40] https://martinfowler.com/bliki/TestCoverage.html

[41] https://github.com/rust-fuzz/afl.rs (08/04/2020)

[42] https://rust-fuzz.github.io/book/afl.html (08/04/2020)

[43] https://www.wireshark.org/docs/wsug_html_chunked/ChIOExportSection.html (08/04/2020)

[44] https://doc.rust-lang.org/nomicon/races.html (20/04/2020)

[45] https://docs.rs/net2/0.2.33/net2/ (21/04/2020)

[46] https://docs.rs/socket2/0.3.12/socket2/ (21/04/2020)

[47] https://doc.rust-lang.org/std/sync/struct.Arc.html (21/04/2020)