The IPbus Protocol

An IP-based control protocol for ATCA/µTCA

IPbus version 2.0 Draft 6 (22nd March 2013)

Robert Frazier, Greg Iles, Marc Magrans de Abril, Dave Newbold, Andrew Rose, David Sankey, Tom Williams

(Authors from v1.2 and previous: Jeremiah Mans, Erich Frahm, Eric Hazen)

1 Introduction

This document describes a simple, reliable, IP-based protocol for controlling hardware devices. It assumes the existence of a virtual bus with 32-bit word addressing and 32-bit data transfer. The choice of 32-bit data width is fixed in this protocol, though the target is free to ignore address or data lines if desired.

1.1 Terminology

- ***IPbus transaction***
  An individual IPbus request or response (e.g. a block read request).
- ***IPbus packet header***
  The first 32 bits of an IPbus packet (containing the packet type and ID fields)
- ***IPbus packet***
  An IPbus packet header plus one or more individual IPbus transactions. These together form the payload of the transport protocol.
- ***IPbus client***
  The software client that generates IPbus transaction requests to control an IPbus target device.
- ***IPbus target***
  The (hardware) device that responds to – and is controlled by – IPbus transaction requests from an IPbus client.
- ***Transport protocol***
  The protocol responsible for transporting the IPbus packet to/from the client/target, e.g. the User Datagram Protocol (UDP).

1.2 Protocol overview

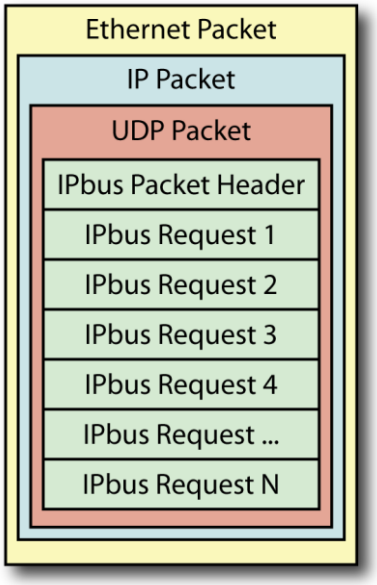
This version of the IPbus protocol has a strong emphasis on simplicity and reliability, as well as ease and compactness of implementation. Where possible, complexity has been pushed into the software client-layer.

All IPbus communication goes through a single port on the target (50001 by default). When sent to or from the target, IPbus transactions are contained within an IPbus packet. In previous versions of IPbus there has been no overall header to an IPbus packet; this has now changed in IPbus v2.0 with the introduction of a single 32-bit header at the front of every IPbus packet, which is necessary to ensure 100% reliable IPbus communication when using a potentially unreliable transport protocol such as UDP. As with previous versions of IPbus, each individual IPbus transaction within the packet also has its own header to describe the content of that particular transaction request/response.

For typical use-cases, with the hardware being controlled on an exclusive private network with simple topology, the recommended transport protocol is UDP. The reason for this choice is to avoid the much greater complexity of implementing a standard reliable transport protocol such as TCP in the firmware of IPbus targets. The recovery of dropped UDP packets is achieved through sequential packet IDs, and the addition of a packet type field in the packet header, which can be used together to form a reliability mechanism.

1.3 IPbus packet structure

The following is an example of an IPbus control packet going from a software (UDP) client to a hardware target. The IPbus-specific information is contained within the payload of the UDP packet, which in turn is wrapped within an IP packet and an Ethernet packet. Regarding the IPbus content of the packet: first there is an IPbus packet header followed by a number of IPbus transaction requests, each of which consists of an IPbus transaction header and – if applicable – its body. The packet returned by the hardware target would follow the same format, except that each IPbus transaction request would be exchanged for the relevant response format. Clearly most efficient usage of the available network bandwidth is made when IPbus packets contain large numbers of request/response transactions. The specifics of the IPbus packet header, IPbus transaction header, and the format of each possible IPbus request/response are detailed in the rest of this document.



It should be noted that in this example, each transaction or set of transactions must fit into a single UDP packet. Note that the maximum size of a standard Ethernet packet (without using jumbo frames) is 1500 bytes; with an IP header of 20 bytes and a UDP header of 8 bytes, this gives the maximum IPbus packet size of 368 32-bit words, or 1472 bytes. Longer block transfers must be split at the software level into individual packets.

1.4 Document Structure

The document is divided in six sections. Section 2 describes the packet header. Section 3, 4, and 5 describes the three different packet types (control, status, and re-send). Finally, section 6 describes the four typical use cases (single and multiple packets in flight, each without packet loss, and using the IPbus reliability mechanism to recover lost packets).

2 IPbus Packet Header

The request and reply of an IPbus packet always begins with a 32-bit header. This header has been added in version 2.0 of the protocol in order to support the reliability mechanism.

The format and content of the packet header is:

	31 24	23 16	15 8	7 0
	Protocol version	Rsvd.	Packet ID (16 bits)	Byte-order qualifier	Packet Type
Word 0	0x2	0	0x0 – 0xffff	0xf	0x0 – 0x2

If an IPbus v2.0 target receives a packet with an invalid packet header, then it silently drops the packet.

With the combination of the *Protocol Version* and the *Byte-order Qualifier* in the most- and least-significant bytes of the packet header respectively, the byte-ordering (i.e. endian-ness) of the 32-bit words within any IPbus packet can be deduced from its header. It is suggested that IPbus target firmware can handle both big and little endian requests, and that each response packet sent back to the client has the same endian-ness as the corresponding request packet. This allows the client software to communicate with targets in the client's native endian-ness, optimizing the efficient usage of standard CPUs on the control computers.

The IPbus protocol defines three *Packet Types*:

Packet Type value	Direction	Meaning
0x0	*Both*	Control packet (i.e. contains IPbus transactions)
0x1	*Both*	Status packet
0x2	*Request*	Re-send request packet
0x3 – 0xf	*n/a*	*Rsvd.*

The main purpose of the *Packet ID* field is to ensure reliable IPbus control communication when using an unreliable transport protocol (e.g. UDP). This is achieved through an IPbus-level reliability mechanism the details of which can be found in the Typical Use Cases section.

3 IPbus Control Packet

An IPbus control packet is the concatenation of a control packet header (*Packet Type* 0x0) and one or many IPbus transactions. Clearly when using a packet-based transport protocol, there is a maximum size to an IPbus control request/response packet – the maximum transmission unit (MTU) of that protocol. As a consequence, it is illegal to make an IPbus control request which would result in a response longer than the path MTU (usually 1500 bytes).

The reply to each IPbus control request packet must start with the same 32-bit packet header (i.e. same *Packet Type* and *Packet ID* values). All requests with non-zero ID received by a target must have consecutive ID values. Otherwise the packet header will be considered invalid and the packet will be silently dropped. A target will always accept control packets with ID value of 0.

The IPbus-level reliability mechanism only works with non-zero packet IDs. For simplicity a packet ID of 0x0 can be used for non-reliable traffic at any time. However it should be noted that the simultaneous use of both forms of IPbus traffic with a single target will disrupt the reliability mechanism.

In the following sub-sections the generic IPbus transaction header is described first, and then the individual transaction types are presented.

3.1 IPbus Transaction Header

Each IPbus request and reply transaction must start with a 32-bit header of the following format:

31 28	27 16	15 8	7 4	3 0
Protocol Version (4 bits)	Transaction ID (12 bits)	Words (8 bits)	Type ID (4 bits)	Info Code (4 bits)

Note that this header is not backwards compatible with previous versions of this protocol (v1.3) – the header format has changed. However, the Protocol Version field is guaranteed consistent across all past and future header definitions. For this version of the protocol (v2.0), the Protocol Version field should always be set to 2.

It should be noted here that each IPbus transaction sent to a target is executed regardless of failures in previous transactions.

The definition of the above fields is as follows:

- *Protocol Version* (four bits at 31 → 28)
  - Protocol version field – should be set to 2 for this version of the protocol.
- *Transaction ID* (twelve bits at 27 → 16)
  - Transaction identification number, so the client/target can track each transaction within a given packet.
  - Note that in order to support the possibility of using jumbo (9kB) Ethernet frames, the transaction ID number cannot be less than 12 bits wide so as to prevent the possibility of the transaction ID wrapping around within a single IPbus packet.
- *Words* (eight bits at 15 → 8)
  - Number of 32-bit words within the addressable memory space of the bus itself that are interacted with, i.e. written/read.
  - Defines read/write size of block reads/writes.

- Reads/writes of size greater than 255 words must be split across two or more IPbus transactions. Having a *Words* field that is only eight bits wide is a necessary trade-off required for keeping the transaction headers only 32 bits in size.

- *Type ID* (four bits at 7 → 4)
  - Defines the type (e.g. read/write) of the IPbus transaction – see the Transaction Types section for the different ID codes.
- *Info Code* (four bits at 3 → 0)
  - Defines the direction and error state of the transaction request/response. All requests (i.e. client to target) must have an Info Code of 0xf. All successful transaction responses must have an Info Code of 0x0. All other values are transaction response error codes (or not yet specified by the protocol).

Info Code	Direction	Meaning
0x0	Response	Request handled successfully by target
0x1	Response	Bad header
0x2	n/a	Rsvd.
0x3	n/a	Rsvd.
0x4	Response	Bus error on read
0x5	Response	Bus error on write
0x6	Response	Bus timeout on read (256 IPbus clock cycles)[1]
0x7	Response	Bus timeout on write (256 IPbus clock cycles)[1]
0x8	n/a	Rsvd.
0x9	n/a	Rsvd.
0xa	n/a	Rsvd.
0xb	n/a	Rsvd.
0xc	n/a	Rsvd.
0xd	n/a	Rsvd.
0xe	n/a	Rsvd.
0xf	Request	Outbound request

3.2 Read transaction (Type ID = 0x0)

Request

	31 24	23 16	15 8	7 0	
Word 0	Version = 2	Transaction ID	Words = READ_SIZE	Type ID = 0	InfoCode = 0xf
Word 1	BASE_ADDRESS				

Response

	31 24	23 16	15 8	7 0	
Word 0	Version = 2	Transaction ID	Words = READ_SIZE	Type ID = 0	InfoCode = 0
Word 1	Data read from BASE_ADDRESS				
Word 2	Data read from BASE_ADDRESS + 1				
…	…				
Word n	Data read from BASE_ADDRESS + (READ_SIZE - 1)				

[1] The usual IPbus clock is 31.25 MHz, resulting in a bus timeout period of approximately 8 µs.

3.3 Non-incrementing read transaction (Type ID = 0x2)

The packet formats for this are identical to the standard read transaction above, except the transaction type ID is 0x2. The non-incrementing read is useful for reading from a FIFO.

3.4 Write transaction (Type ID = 0x1)

Request

	31 24	23 16	15 8	7 0	
Word 0	Version = 2	Transaction ID	Words = WRITE_SIZE	Type ID = 1	InfoCode = 0xf
Word 1	BASE_ADDRESS				
Word 2	Data for BASE_ADDRESS				
Word 3	Data For BASE_ADDRESS + 1				
…	…				
Word n	Data for BASE_ADDRESS + (WRITE_SIZE -1)				

Response

	31 24	23 16	15 8	7 0	
Word 0	Version = 2	Transaction ID	Words = WRITE_SIZE	Type ID = 1	InfoCode = 0

3.5 Non-incrementing write transaction (Type ID = 0x3)

The packet formats for this are identical to the standard write transaction above, except the transaction type ID is 0x3. The non-incrementing write is useful for writing to a FIFO.

3.6 Read/Modify/Write bits (RMWbits) transaction (Type ID = 0x4)

The RMWbits transaction is useful for setting or clearing bits. Only single-word (32-bit) RMWbits transactions are defined. The algorithm performed is the following:

X <= (X & A) | B

where ***X*** is the existing value, ***A*** is the AND term, and ***B*** is the OR term.

Using the RMWbits transaction, it is possible to perform efficient “masked writes” on a 32-bit register using a single transaction.

Request

	31 24	23 16	15 8	7 0	
Word 0	Version = 2	Transaction ID	Words = 1	Type ID = 4	InfoCode = 0xf
Word 1	BASE_ADDRESS				
Word 2	AND term				
Word 3	OR term				

Response

	31 24	23 16	15 8	7 0	
Word 0	Version = 2	Transaction ID	Words = 1	Type ID = 4	InfoCode = 0
Word 1	Content of BASE_ADDRESS as read before the modifty/write				

3.7 Read/Modify/Write sum (RMWsum) transaction (Type ID = 0x5)

The RMWsum transaction is useful for adding values to a register (or subtracting, using two's complement). The algorithm performed is the following:

X <= (X + A)

where ***X*** is the existing value, and ***A*** is the ADDEND.

Only single-word (32-bit) RMWsum transactions are defined.

Request

	31 24	23 16	15 8	7 0	
Word 0	Version = 2	Transaction ID	Words = 1	Type ID = 5	InfoCode = 0xf
Word 1	BASE_ADDRESS				
Word 2	ADDEND				

Response

	31 24	23 16	15 8	7 0	
Word 0	Version = 2	Transaction ID	Words = 1	Type ID = 5	InfoCode = 0
Word 1	Content of BASE_ADDRESS as read before the summation				

4 IPbus Status Packet

The status packet is new in this version of the protocol, and is necessary for the reliability mechanism, as well as being useful for debugging the target's recent activity.

Request

	31 24	23 16	15 8	7 0
Word 0	Version = 2 / Rsvd. = 0	Packet ID = 0		Byte-order qualifier = 0xf / Pkt. Type = 1
Word 1	Rsvd. = 0			
Word 2	Rsvd. = 0			
Word 3	Rsvd. = 0			
Word 4	Rsvd. = 0			
Word 5	Rsvd. = 0			
Word 6	Rsvd. = 0			
Word 7	Rsvd. = 0			
Word 8	Rsvd. = 0			
Word 9	Rsvd. = 0			
Word 10	Rsvd. = 0			
Word 11	Rsvd. = 0			
Word 12	Rsvd. = 0			
Word 13	Rsvd. = 0			
Word 14	Rsvd. = 0			
Word 15	Rsvd. = 0			

Response

	31 24	23 16	15 8	7 0
Word 0	Version = 2 / Rsvd. = 0	Packet ID = 0		Byte-order qualifier = 0xf / Pkt Type = 1
Word 1	MTU in bytes			
Word 2	nResponseBuffers (the maximum packet overlap the firmware will support)			
Word 3	Next expected IPbus packet header			
Word 4	Incoming traffic history 0 (oldest)	Incoming traffic history 1	Incoming traffic history 2	Incoming traffic history 3
Word 5	Incoming traffic history 4	Incoming traffic history 5	Incoming traffic history 6	Incoming traffic history 7
Word 6	Incoming traffic history 8	Incoming traffic history 9	Incoming traffic history 10	Incoming traffic history 11
Word 7	Incoming traffic history 12	Incoming traffic history 13	Incoming traffic history 14	Incoming traffic history 15 (newest)
Word 8	Control packet history: received IPbus control packet header 0 (oldest)			
Word 9	Control packet history: received IPbus control packet header 1			
Word 10	Control packet history: received IPbus control packet header 2			
Word 11	Control packet history: received IPbus control packet header 3 (newest)			
Word 12	Control packet history: outgoing IPbus control packet header 0 (oldest)			
Word 13	Control packet history: outgoing IPbus control packet header 1			
Word 14	Control packet history: outgoing IPbus control packet header 2			
Word 15	Control packet history: outgoing IPbus control packet header 3 (newest)			

Several fields from this status response are explained below:

- *MTU in bytes* is the maximum transmission unit that the firmware supports for IPbus - i.e. the maximum length of IPbus packet (in bytes) that the firmware can handle.
- The *Incoming traffic history* bytes give basic details about the last 16 Ethernet packets sent to the target board. Each entry is constructed like so:

  Bit 7 Failed CRC flag

Bit 6 Dropped on receipt (i.e. no incoming buffer space available)
Bit 5 Rsvd
Bit 4 Rsvd
Bit 3:0 Event Type Encoding

where the *Event Type Encoding* values have the following meanings:

0 Hard reset of target
(N.B. A hard reset will have cleared all the previous status history)
1 IPbus reset – i.e. an internal reset of IPbus firmware modules
(N.B. An IPbus reset does not clear any previous status history)
2 IPbus control request packet
3 IPbus status request packet
4 IPbus resend request packet
5 Unrecognised UDP packet to IPbus port (i.e. invalid header)
6 Valid ping
7 Valid ARP
8 – 14 Rsvd
15 Any other non-broadcast Ethernet packet

- The last eight 32-bit words in the packet then contain the headers of the last four IPbus packets received at and sent from the control port.

Within this status response packet, the byte-ordering within words 0 to 7 is always big-endian. For each of the last 8 words the byte-ordering is the same as the byte-ordering of the original response / request packet.

5 IPbus Re-send Packet

The re-send request packet is also a new addition of this protocol version and it is denoted by a *Packet Type* value of 0x3 in the packet header. This packet type is necessary for the reliability mechanism, and its purpose is to trigger a re-send of one of the target's recent outbound control packets.

To trigger a re-send of a control response packet with ID *N*, the following re-send request packet must be send:

	31 – 24	23 – 16	15 – 8	7 – 0	
Word 0	Version = 2	Rsvd. = 0	Packet ID = *N*	Byte-order qualifier = 0xf	Pkt. Type = 2

If a packet with the requested ID is in the target's outbound buffer, then the target will re-send that response control packet to the same client as it was originally sent to; otherwise, if there isn't any packet with this ID in the target's outbound buffer then no packet is sent to client.

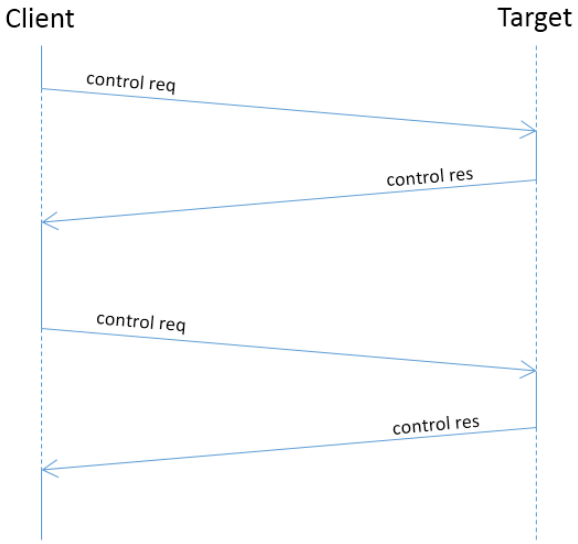
6 Typical Use Cases

In this part of the document, sequence diagrams outlining the client-target interactions are shown for a few typical use case scenarios.

6.1 Fault-free Single Packet in Flight

The simplest use case is when the transport protocol does not fail and there is only one packet in flight. In this case, only control packet request and responses are sent over the network.

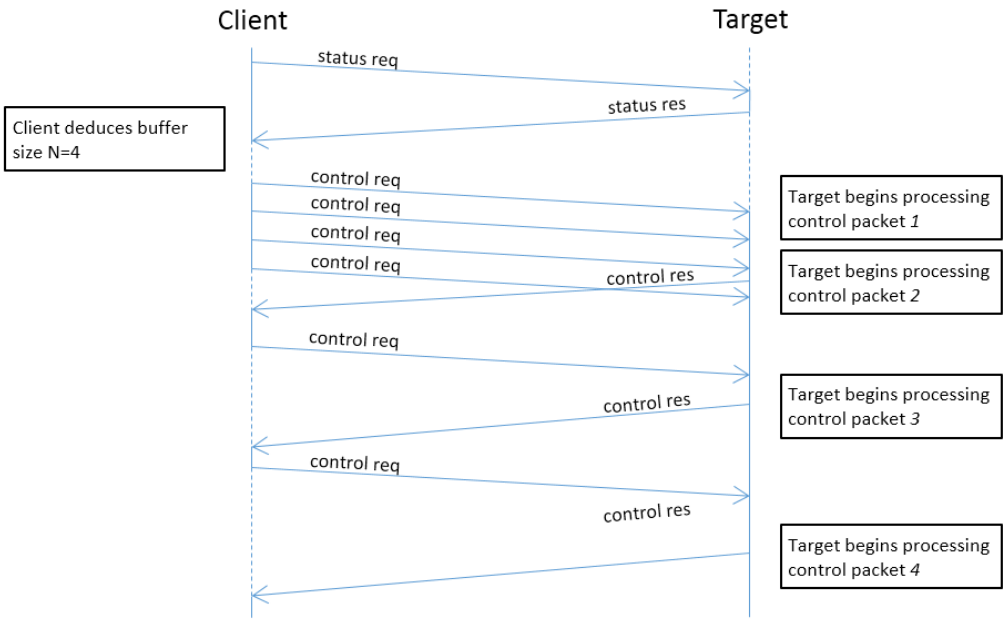
Notice that the diagram does not show the Packet ID as they are not relevant in this case.



6.2 Fault-free Multiple Packets in flight

In order to increase the bandwidth usage, it is useful for a client to be able to send multiple control request packets to an IPbus target before any of the corresponding response packets have been received; this mode of operation is referred to as multiple packets in flight. Clearly, in order for N packets to be sent to a target before their response is sent back, the target must have a buffer that can store N incoming IPbus control packets.

The figure below shows a sequence diagram of multiple packets in flight when the buffer length N is 4. Note that before it starts sending packets, the client has to send a status request to know the inbound buffer length N.



6.3 Reliability Mechanism for Single Packet in Flight

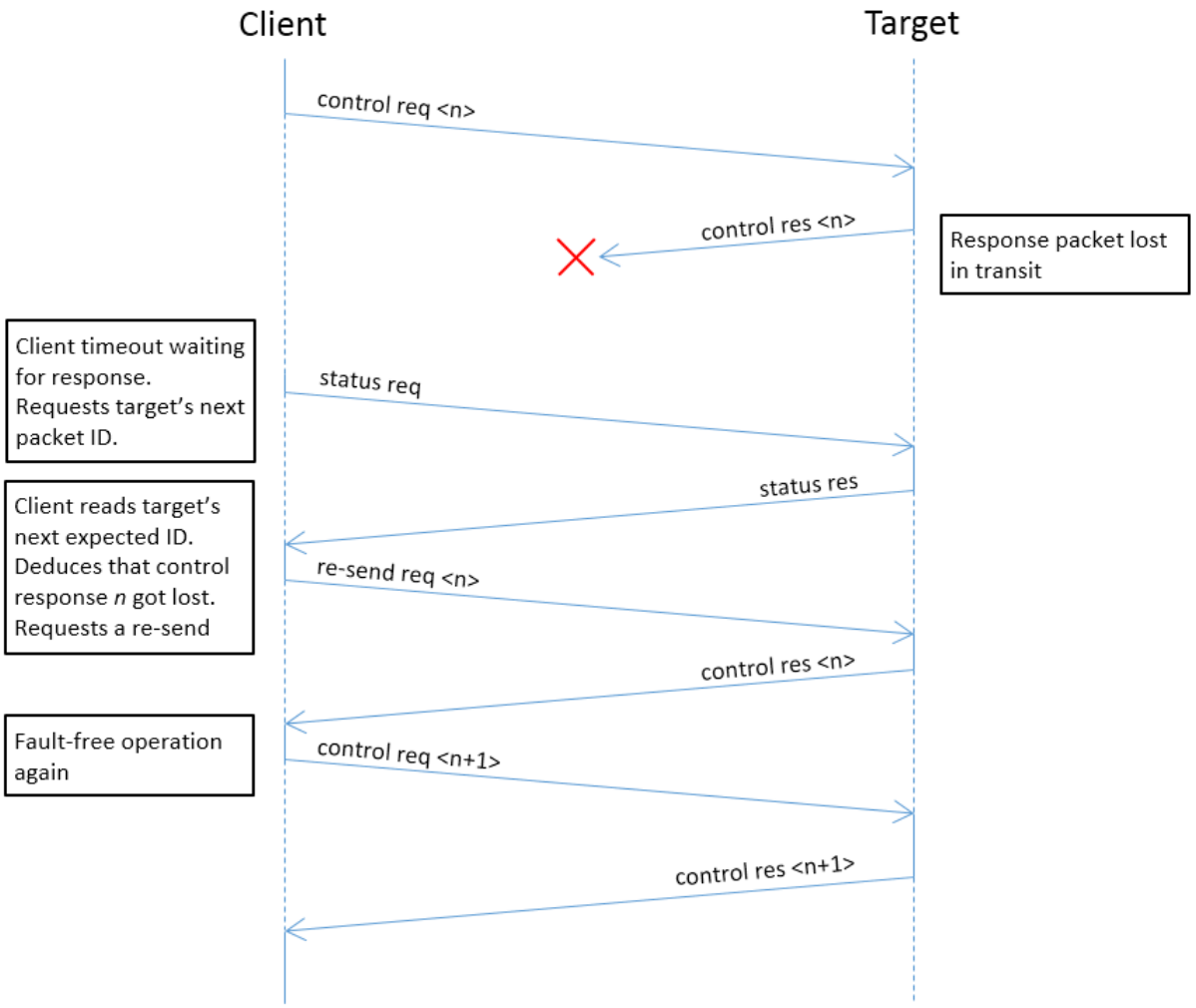
The IPbus packet-level reliability mechanism is designed to recover from the loss of control packets when using an unreliable transport protocol such as UDP.

The reliability mechanism is achieved through a combination of three features:

- The client and target enforce consecutive Packet IDs on consecutive control packets.
- The client can determine the next packet ID that the target is expecting by sending a status packet request.
- The client can issue a re-send request packet to ask for the target to re-send its last control packet.

Notice that the reliability mechanism will only work correctly when there is a single client communicating with the target (i.e. either a single control application, or multiple control applications whose traffic is routed via a single ControlHub instance). Additionally the client and target must have a buffer to store their last outbound control packet in case it gets lost in transit.

The following diagram is an example of packet response loss recovery through the IPbus reliability mechanism.



6.4 Reliability Mechanism for Multiple Packets in Flight

The use case in which the reliability mechanism is applied to multiple packets in flight is just a combination of the single packet in flight reliability mechanism, and fault-free operation with multiple packets in flight. Clearly, in order for N packets to be sent reliably to a target before their response is sent back the target must have incoming and outgoing buffers that can store the N most recent control packets (requests and responses, respectively).

Additionally, it should be noted that as with the single packet in flight case, the reliability mechanism will only work if there is a single client. The figure below shows an example of packet loss recovery through the IPbus reliability mechanism with multiple packets in flight when the buffer length N is 4. Note that the client has to send a status request initially in order to know the inbound and outbound buffer size.