Latency **M**easurement **P**rotocol

Custom protocol specifications

Revision 2.0

Contents

1. Introduction and basic concepts

LaMP (**L**atency **M**easurement **P**rotocol) is a custom application layer protocol, which can be placed at layer 7 of the ISO/OSI model.

This custom protocol has been specifically designed to support latency and RTT measurements using a client-server paradigm and trying to make it as flexible as possible, making it independent on what it is behind it in the protocol stack.

In fact, LaMP is meant to be encapsulated in any protocol, even directly inside raw Ethernet or 802.11 packets, to assess the latency performance of different devices using different protocols. The flexible approach also allows the user to transmit additional data and perform other time-related custom measurements.

Being it a client-server based protocol, it always require at least a LaMP client running on one device and a LaMP server running on another device; then, thanks to specific header fields, including the identification and sequence number, the protocol implementation should be able to handle also the case of multiple clients and servers launched on the same machine.

Each client connecting to a specific server instance is part of a *session*, which is a single latency measurement session, characterized by a very specific identification number, from 0 to 65535. Client and servers are then referred, in this document, as *entities*.

In each session, packets are exchanged between two *entities* (one client and one server).

When a session is going to terminate, at least one *entity* belonging to a *session* should prepare a *report*, containing at least the data listed in section 3.6, and present it to the user, store it locally or send it to other processes.

Due to its latency measurement characteristics, the LaMP header always contain two fields to store a microsecond-precision timestamp, which is managed by the *entities* participating in a measurement *session*. Typically, a send timestamp should always be compared with a receive one, making, in the most basic case, a subtraction to compute the latency between the two.

The exact instant in which a timestamp is placed inside a LaMP packet is left to the user, depending on the kind of latency he or she wants to measure. Normally, the timestamp should be placed inside the packet (or, in any case, should be retrieved) in the last possible instant before sending LaMP data, in order to try to reduce the latency contribution due to the user application.

The next sections will cover more in details, trying to be short, how LaMP should work.

This protocol has been developed in **Politecnico di Torino**, at the **DET** department (**department of electronics and telecommunications)** and its specifications are always meant to be **open** and **available**.

Revision 2.0 introduces new features to LaMP, including the follow-up mechanism (section 3.6), and fixes some typographical errors inside the document.

2. Packet format

The LaMP packet is composed by a *header* and a *payload*. While the *header* length is fixed to 24B, the *payload* is variable in length, ranging from 0 B to 65535 B. Moreover, the payload is not mandatory (except for the INIT packets, see chapter 3) and can also be omitted (i.e. when "**Length or packet type**" is set to "0").

The *header* is divided in 3 blocks of 8 B. The first block contains all the information related to the packet type, sequence number, session id, and so on. The second and the third blocks are instead reserved respectively for seconds and for microseconds timestamps. The entire packet format, with some hints for the field usage, can be seen in the following table.

	Byte		0	1	2	3	4	5	6	7
		Bit	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31	32 33 34 35 36 37 38 39	40 41 42 43 44 45 46 47	48 49 50 51 52 53 54 55	56 57 58 59 60 61 62 63
Header	0	0	Reserved	Control 0xA ¦ Pkt Type	LaMP ID		Sequence No.		Length or packet type	
	8	64	Sec Timestamp							
	16	128	uSec Timestamp							
Payload (optional)	24	172	Payload (optional) – up to 65535B							
	…	…								
	65552	524416								

Fields Description

Byte	Length (B)	Name	Description	Value	
0	**1**	**Reserved**	**Reserved field: it is used to determine if the packet is a LaMP packet**	0xAA	
1	**1**	**Control**	**Control field: the first 4 bits should always be set to 0xA, while the second 4 bits are used to encode the packet type**	0xA	0x0: PINGLIKE_REQ 0x1: PINGLIKE_REPLY 0x2: PINGLIKE_ENDREQ 0x3: PINGLIKE_ENDREPLY 0x4: UNIDIR_CONTINUE 0x5: UNIDIR_STOP 0x6: REPORT 0x7: ACK 0x8: INIT 0x9: PINGLIKE_REQ_TLESS 0xA: PINGLIKE_REPLY_TLESS 0xB: PINGLIKE_ENDREQ_TLESS 0xC: PINGLIKE_ENDREPLY_TLESS 0xD: FOLLOWUP_CTRL 0xE: FOLLOWUP_DATA 0xF: *RESERVED (Future use)*
[2,3]	**2**	**LaMP ID**	**Identification field: it is used to uniquely identify the LaMP session**	Uniform(0x0000,0xFFFF)	
[4,5]	**2**	**Sequence No.**	**Sequence field: it is used to store cyclically increasing sequence numbers, up to 0xFFFF (65535). It is used to identify packets loss, out-of-order packets and to associate replies with requests**	Increasing from 0x0000 to 0xFFFF	
[6,7]	**2**	**Length or packet type**	**Length or packet type field: it is used to store the (optional) payload length, up to 65535B. If the packet type is INIT, it stores the type of connection that should be established (pinglike or unidirectional), if it is FOLLOWUP_CTRL, it stores the type of follow-up control packet**	If packet type == INIT	INIT type (*0x0001* or *0x0002*)
				If packet type == FOLLOWUP_CTRL	Follow-up control type
				If packet type == *any other*	Maximum value: 0xFFFF
[8,15]	**8**	**Sec Timestamp**	**Sec Timestamp field: it is used to store the seconds of the current timestamp**	Maximum value: 0xFFFFFFFFFFFFFFFF In case of TLESS it is set to 0	
[16,23]	**8**	**uSec Timestamp**	**uSec Timestamp field: it is used to store the microseconds of the current timestamp**	Maximum value: 0xFFFFFFFFFFFFFFFF In case of TLESS it is set to 0	
[24,65559]	**0 - 65535**	**Payload (optional)**	**Payload field: it is used to store the payload of the packet. This field is optional**	User defined payload	

The first byte is **reserved** and is always set to 0xAA. This allows the user to determine whether a packet is a LaMP packet.

The second byte contains the **control** field: this information is used to identify the type of LaMP packet that is being transmitted/received. This byte is subdivided in 2 sections of 4 bits: the first is always set to 0xA, as an extension of the reserved field, while the latter encodes the packet type.

The third and the fourth bytes are used to identify the **LaMP session**. Since this protocol is meant to be used in a client-server paradigm, having a session identifier allows the application using it to discriminate the packets belonging to different instances and to open multiple sessions on the same terminal.

The fifth and the sixth bytes contain the **sequence number** of the packet, that is constantly increased (per-session) and that helps the application to detect out-of-order packets or packet losses. Sequence numbers are also used to associate the replies and the corresponding follow-up messages, when a follow-up mechanism is used.

The last two bytes of the first block are used to determine the **payload length** or, in case the packet is of type INIT, they define the **type of connection** that will be established (ping-like, i.e. bidirectional, or unidirectional).

The second and the third block contain the **64bits timestamps** for second and microseconds. Since the LaMP protocol can also be used in *timestampless* fashion, these bytes may be set to 0.

The payload is *user defined*, meaning that it can be filled up with information depending on the user needs, including encapsulating other protocols inside LaMP.

The next sections provide more detailed information about each field of the LaMP packet.

3. Rules

3.1. Packet types and usage

The LaMP protocol foresees different packet types, which are distinguished by the 4-bit value "**Pkt Type**" inside the "**Control**" field of the protocol header.

The following table lists all the currently available types and their usage.

For each packet type, the sending entity (client, server or both) is listed.

Packet type ***Entity***	**Short name**	**Control field value**	**Usage**
Ping-like (bidirectional) request ***Client***	PINGLIKE_REQ	*0xA0*	This packet represents a typical ping-like request, in which the client sends a packet to the server and the server shall reply with a corresponding ping-like reply, like what happens in the ICMP Echo Request/Reply mechanism. The client is responsible for the insertion of the timestamp inside the request packet and to ensure that it is as much accurate as possible. At each transmission, the sequence number must be increased.
Ping-like (bidirectional) reply ***Server***	PINGLIKE_REPLY	*0xA1*	This packet should be sent by the server back to the client, upon receiving a PINGLIKE_REQ. The content of the packet (including the timestamp) must be exactly the same as the request received by the client, with the exception of the control field, which should be now set to *0xA1*. The payload should be typically the same, but it is not mandatory, and can be changed to transmit user defined data to the client or to perform asymmetric measurements with respect to the payload size.
Ping-like (bidirectional) end request ***Client***	PINGLIKE_ENDREQ	*0xA2*	When the client is about to send the last request of the current session, it shall send, instead of a normal PINGLIKE_REQ, a final request packet, in order to make the server aware of the fact that this will be the last packet of the session and let it perform memory cleanup, reporting, and other final operations. The client is responsible for the insertion of the timestamp inside the request packet and to ensure that it is as much accurate as possible.

Ping-like (bidirectional) end reply ***Server***	PINGLIKE_ENDREPLY	*0xA3*	When a PINGLIKE_ENDREQ is received by the server, it should reply with a PINGLIKE_ENDREPLY instead of a normal one, respecting, however, all the rules for normal server replies.
Unidirectional continue ***Client***	UNIDIR_CONTINUE	*0xA4*	This packet must be used by a client to send requests to a server when working in the unidirectional mode. Like in PINGLIKE_(END)REQ, a precise timestamp must be set inside the header, and will be used by the server to compute the delta between the timestamp extracted from the packet and its own timestamp, which can be obtained, for instance, as soon as the UNIDIR_CONTINUE packet is received. As the mode is now unidirectional, no reply is expected from the server. This packet also indicates that there will be more packets after the current one and the server should be ready to process them. At each transmission, the sequence number must be increased.
Unidirectional stop ***Server***	UNIDIR_STOP	*0xA5*	This packet must be used by a client to send requests to a server when working in the unidirectional mode. Like in PINGLIKE_(END)REQ, a precise timestamp must be set inside the header, and will be used by the server to compute the delta between the timestamp extracted from the packet and its own timestamp, which can be obtained, for instance, as soon as the UNIDIR_STOP packet is received. As the mode is now unidirectional, no reply is expected from the server. This packet also indicates that the session is going to be terminated and that no unidirectional packets will follow the current one.
Report data ***Server(/Client)***	REPORT	*0xA6*	This packet contains, inside its payload, user-formatted report data, which is exchanged between the client and server, containing statistics about the measured latency values and, possibly, also about packet loss and out of order packets. Typically, it can be used by the server to send back report data to the client, for instance when working in unidirectional (experimental) mode, to let the client

			print all the report data even if the computation is performed by the server. This is not mandatory, though. This packet must have a payload length different than 0 and it must contain report data inside the payload. The format of the report data, which is used to parse it when a REPORT is received, is up to the user. The timestamp is typically not required (i.e. should be set to 0) but it can be inserted by the user if needed.
Acknowledgment ***Client/Server***	ACK	*0xA7*	This is a typical general acknowledgement packet, which can be used by the client or by the server to acknowledge the reception of a certain packet. It should be used to confirm the reception of INIT and REPORT packets. Typically an ACK does not contain any payload (i.e. "**Length or packet type**" should be equal to 0), but the user can include and manage a custom payload and this should not be forbidden by any protocol implementation, in order to make it as much flexible as possible. The timestamp is typically not required (i.e. should be set to 0) but it can be inserted by the user if needed.
Connection initialization ***Client***	INIT	*0xA8*	This packet should be sent by any client to perform an initial handshake with a server, before starting any session, and let the server set few session parameters, such as the ID of the packets it will accept. This can be performed thanks to the header fields contained inside the special INIT packet. INIT packet must be followed by an ACK, sent by the server, to let the client know that the server is ready. After the INIT packet has been processed by the server and an ACK has been received by the client, the session can start. INIT packets must have no payload and the "**Length or packet type**" set to *0x0001* to initiate a ping-like (bidirectional) session and to *0x0002* to initiate an unidirectional session. See sections 3.4 and 3.5 for more detailed information.

			The timestamp is typically not required (i.e. should be set to 0) but it can be inserted by the user if needed.
Ping-like (bidirectional) request, timestampless ***Client***	PINGLIKE_REQ_TLESS	*0xA9*	This packet type works exactly like PINGLIKE_REQ, but it does not carry any timestamp (i.e. both timestamp fields must be set to 0). In this case the client will be responsible for keeping the send timestamp inside a proper data structure, instead of placing it inside the packet, in order to store it and compare it with the receive timestamp, which can be obtained, for instance, as soon as a timestampless reply is received from the server.
Ping-like (bidirectional) reply, timestampless ***Server***	PINGLIKE_REPLY_TLESS	*0xAA*	This packet type works exactly like PINGLIKE_REPLY, but it does not carry any timestamp (i.e. both timestamp fields must be set to 0).
Ping-like (bidirectional) end request, timestampless ***Client***	PINGLIKE_ENDREQ_TLESS	*0xAB*	This packet type works exactly like PINGLIKE_ENDREQ, but it does not carry any timestamp (i.e. both timestamp fields must be set to 0).
Ping-like (bidirectional) end reply, timestampless ***Server***	PINGLIKE_ENDREPLY_TLESS	*0xAC*	This packet type works exactly like PINGLIKE_ENDREPLY, but it does not carry any timestamp (i.e. both timestamp fields must be set to 0).
Follow-up control ***Client/Server***	FOLLOWUP_CTRL	*0xAD*	This packet embeds the follow-up mechanism control data. These packets must have no payload and the "**Length or packet type**" header field set to a different value depending on the control data to transmit. As a client wants to use the follow-up mechanism, in which each reply (ping-like) or unidirectional message (unidirectional) is followed by a follow-up data packet carrying additional time information (to increase the latency measurement accuracy), it should issue a **follow-up request**, setting this field to *0x00__*. The last byte, from *0x__00* to *0x__FF*, is used to specify a type of request (see section 3.6).

			Then, the server can accept or deny the request, by sending back another follow-up control packet with “**Length or packet type**” set to: • *0x0100* for **deny** • *0x0200* for **accept** • *0xFFFF* when a **badly formatted or unknown request** (according to the type) is received In case the request is accepted, the session can start with the follow-up mechanism, otherwise no follow-up should be used by both the client and the server.
Follow-up data ***Server/Client***	FOLLOWUP_DATA	*0xAE*	This packet embeds the follow-up mechanism timestamp data. This packet, when the follow-up mechanism is active and it has been negotiated by the client and the server, should carry timestamp data coming from the sending entity, depending on the connection type: • Ping-like mode: the best possible estimate of the processing time spent by the server when managing a request and preparing/sending the corresponding reply; the embedded timestamp should represent a time delta instead of a single timestamp • Unidirectional mode: the best possible estimate of the transmission timestamp, related to the packet immediately preceding the follow-up message, when it is possible to retrieve a send timestamp which is more precise than any timestamp which can be embedded inside UNIDIR_CONTINUE or UNIDIR_STOP packets.
Reserved **-**	-	*0xAF*	This control field value is reserved for future use and it should never be used in any LaMP packet.

3.2. Header fields management

The following table lists all the header fields for each packet type, describing how they shall be managed according to the current LaMP packet.

Packet type	Res	Ctrl	ID	Sequence number	Len or packet type	Sec Timestamp	uSec Timestamp	Payload
PINGLIKE_REQ	0xAA	0xA0	Unique random ID for each session; must be same as the one sent within the INIT packet	Increasing (by 1 or with any user-defined rule)	Length of the payload	Required	Required	Optional
PINGLIKE_REPLY	0xAA	0xA1	Unique random ID for each session, same as request	Same as request	Length of the payload	Same as request	Same as request	Optional
PINGLIKE_ENDREQ	0xAA	0xA2	Same ID as the previous PINGLIKE_REQ	Increasing (by 1 or with any user-defined rule)	Length of the payload	Required	Required	Optional
PINGLIKE_ENDREPLY	0xAA	0xA3	Unique random ID for each session, same as end request	Same as end request	Length of the payload	Same as end request	Same as end request	Optional
UNIDIR_CONTINUE	0xAA	0xA4	Unique random ID for each session; must be same as the one sent within the INIT packet	Increasing (by 1 or with any user-defined rule)	Length of the payload	Required	Required	Optional
UNIDIR_STOP	0xAA	0xA5	Same ID as the previous UNIDIR_CONTINUE	Increasing (by 1 or with any user-defined rule)	Length of the payload	Required	Required	Optional
REPORT	0xAA	0xA6	Same ID as the one used in the current session	Starting from 0, then increased by 1 at each retransmission attempt	Length of the payload. Must be different than 0.	Optional (typically not required)	Optional (typically not required)	Required and containing report data (i.e. statistics)
ACK	0xAA	0xA7	Same ID as the one used in the current session	Starting from 0, then increased by 1 at each retransmission attempt	Typically 0; if a payload is really needed, it contains it length.	Optional (typically not required)	Optional (typically not required)	Optional (typically not required)
INIT	0xAA	0xA8	Unique random ID for each session	Starting from 0, then increased by 1 at each retransmission attempt	Connection INIT type: *0x0001* for ping-like, *0x0002* for unidirectional	Optional (typically not required)	Optional (typically not required)	No
PINGLIKE_REQ _TLESS	0xAA	0xA9	Unique random ID for each session; must be same as the one sent within the INIT packet	Increasing (by 1 or with any user-defined rule)	Length of the payload	No (set to 0)	No (set to 0)	Optional
PINGLIKE_REPLY _TLESS	0xAA	0xAA	Unique random ID for each session, same as request	Same as request	Length of the payload	No (set to 0)	No (set to 0)	Optional

PINGLIKE_ENDREQ _TLESS	0xAA	0xAB	Same ID as the previous PINGLIKE_REQ _TLESS	Increasing (by 1 or with any user-defined rule)	Length of the payload	No (set to 0)	No (set to 0)	Optional
PINGLIKE_ENDREPLY _TLESS	0xAA	0xAC	Unique random ID for each session, same as end request	Same as end request	Length of the payload	No (set to 0)	No (set to 0)	Optional
FOLLOWUP_CTRL	0xAA	0xAD	Same ID as the one used in the current session	Starting from 0, then increased by 1 at each retransmission attempt	Follow-up control message type: *0x00*__ for request (the last byte is used to specify a type of request) *0x0100* for (reply) deny *0x0200* for (reply) accept *0xFFFF* for bad or unknown request received	No (set to 0)	No (set to 0)	No
FOLLOWUP_DATA	0XAA	0xAE	Same ID as the one used in the current session	Same as the reply to which the follow-up packet is referred	Length of the payload	Required: estimate of internal application time to be subtracted from the computed latency/RTT.	Required: estimate of internal application time to be subtracted from the computed latency/RTT.	Optional (typically not required)
(Reserved)	0xAA	0xAF	-	-	-	-	-	-

3.3. Sequence numbers

Sequence numbers shall be used:

- In INIT, ACK, REPORT and FOLLOWUP_CTRL packets, always starting from 0, to identify each retransmission attempt. In this case they should always be increased by 1, until a user-defined maximum number of retransmissions is reached.
- In all the other packets: they can start from any user defined value, and increased by 1 or following any increasing user defined rule. They are used, as highlighted before, to associate replies with requests and, possibly, to compute a packet loss and out-of-order count. Typically, the suggestion is to always start from 0 and increase them by 1, but no rule is strictly defined in order to maintain flexibility within the protocol specifications. However, if no well-tested and well-defined rule can be defined and documented, the user shall start from 0 and increase the sequence numbers by 1.

Sequence number are stored in a 16 bit field of the LaMP header, thus their maximum value is 65535 (*0xFFFF*). They shall be cyclically increased when such a maximum value is reached, starting back from 0.

3.4. Ping-like operating mode

When operating in **ping-like mode** (**RTT** measurements) the transmission and reception of latency measurement packets should happen in a similar way to the ICMP Echo Request/Reply mechanism, with few additional operations, respecting all the rules defined in the previous sections.

I. The connection is always initiated by the client, which should send an INIT packet to the server. The INIT packet may contain a timestamp in order to perform any time initialization operation from the server side, but it is not required and in the most common cases can be omitted (i.e. leaving it to 0, for what concerns both the **Sec** and **uSec** fields).
   The initialization packet shall also contain a random ID, stored inside the 16 bit field of the LaMP header (it can be any number from 0 to 65535) and kept as it is until the session is terminated.
   The "**Length or packet type**" field should be set to *0x0001*.

II. The server, upon receiving the INIT packet, should use its content to set the ID and the session type (i.e. *ping-like*). Then, the server should accept only packets containing that ID and using the mode specified inside the "**Length or packet type**" field of the INIT packet. The server can optionally gather statistics about the INIT retransmission attempts, using the sequence numbers, and perform other user defined time related operations, in case a timestamp was inserted inside the packet.

III. After properly processing the INIT packet, the server should send back an ACK packet to the client, which will be informed about the server being ready to start the current measurement session.

IV. In the mean time, the client should repeat the transmission of the INIT packet, using a transmission interval defined by the user, until the ACK is received from the server or until a certain number of attempts is reached. In the last case, the client should terminate the current session. If no packets are lost, a single INIT-ACK transmission should occur, i.e. in ideal channel/link conditions, with no losses, the server should be fast enough to reply to the client before the first retransmission occurs and the client should not set a period which is too short, causing unnecessary retransmissions towards the server.

V. After the INIT-ACK procedure is completed, the client can start sending normal or timestampless packets to the server (PINGLIKE_REQ or PINGLIKE_REQ_TLESS):
   1. In case normal packets are sent (PINGLIKE_REQ), the client should prepare them according to the rules specified before. It will set an initial sequence number, which will be then increased by 1 (or using a user defined rule) for each request transmission and optionally fill the payload.
      If a payload is used, the "**Length or packet type**" field should be different than 0 and equal to the number of bytes contained inside the payload. The payload can contain random data, which will not be parsed neither by the client (when receiving the corresponding reply) nor by the server, in order to perform latency measurements with different payload sizes, or it can contain meaningful data which can be parsed by the client (upon reception of the reply) or by the server. This allows user defined per-packet data to be easily transmitted between entities using LaMP.
      It is mandatory to set a timestamp inside the packet. The timestamp can be obtained and placed inside the header at any time, using system calls such as *gettimeofday()*, under Linux, depending on the kind of latency measurement the user wants to perform.
      Typically, the timestamp should be set in the last possible instant before sending the packet, in order to reduce the application latency as much as possible.
   2. In case timestampless packets are sent (PINGLIKE_REQ_TLESS), all the rules described in (1) apply, but for the insertion of the timestamp, which must be left to all zeros.

In this case the client will be responsible for storing the transmit timestamp in a local data structure.

This data structure should then be used to retrieve the transmit timestamp corresponding to the received timestampless replies, and, together with a receive timestamp obtained by the client upon reception of the replies (for instance, under Linux, by means of *gettimeofday()* or CSMG ancillary data), to compute the latency values by subtracting the obtained received timestamp with the stored transmit timestamp.

This mode has been inserted in order to support hardware transmit timestamps under Linux, which are returned, when supported, as ancillary data, and thus cannot be easily embedded inside the requests which are sent to the server.

VI. The server shall reply, after the INIT-ACK procedure is complete, to all the client requests with the correct ID and mode (i.e. in case of ping-like operations it shall accept all the PINGLIKE_REQ, PINGLIKE_ENDREQ, PINGLIKE_REQ_TLESS and PINGLIKE_ENDREQ_TLESS packets, provided that the ID is correct).

The replies shall replicate the same exact content of the requests (including the received timestamp, set by the client), but for the packet type, which should be set to PINGLIKE_REPLY or PINGLIKE_REPLY_TLESS. The payload, in standard latency measurements with a defined payload size, should remain the same, but it is not mandatory and it can change when the user wants to transmit additional user defined data to the client or when performing (but it is a quite uncommon situation) asymmetric ping-like latency measurements with respect to the payload size.

VII. As the client receives a reply from the server, it should use it to gather statistics, perform time related operations and possibly parse the payload, if it contains meaningful data.

The most important operation a client should perform is the latency computation. In case standard packets are used, the client should extract the timestamp stored inside the reply packet (which reflects the timestamp the client set inside the corresponding request) and compare it with the receive timestamp, which can be obtained depending on the user needs. Under Linux, this receive timestamp can be obtained, for instance, with *gettimeofday()*, called just after recognizing that the current packet, received through a *socket*, is of interest (i.e. if it is LaMP, it is a reply and it has the correct ID) or by means of ancillary data.

In case timestampless packets are used instead, the client, as stated before, is responsible for keeping the send timestamps (which can be obtained, for instance, by means of ancillary data, when using Linux) and compare them with the correct receive timestamps, following any user defined policy, which should be documented together with the application using LaMP.

VIII. When the client is about to terminate its session, it should send the last request packet as an end request one, respecting all the rules defined before for client requests and setting its type to PINGLIKE_ENDREQ, for normal packets, or PINGLIKE_ENDREQ_TLESS, for timestampless packets. This will inform the server about the intention to terminate the current session, letting it perform memory cleanup, reporting, computation and other operations.

IX. Upon reception of a PINGLIKE_ENDREQ or PINGLIKE_ENDREQ_TLESS packet, the server should reply with a PINGLIKE_ENDREPLY or PINGLIKE_ENDREPLY_TLESS respectively, following all the rules described in (VI). It can then consider the session as terminated and can initiate the execution of the final memory cleanup and reporting operations.

X. When receiving an end reply packet, the client can consider the session as terminated too and can initiate the execution of the final memory cleanup and reporting operations.

XI. Optionally, the server and/or the client can start exchanging report data and gather statistics, to be used inside the application or to be displayed to the user.

Each data exchange should occur in the following way:

1. The report data sender (client or server) should start sending the report data, inside the payload of REPORT packets, at periodic intervals, defined by the user, until a

certain number of retransmissions is reached (the first is the real transmission, the other ones are retransmissions of the same packet, with an increased sequence number).

2. The report data receiver (server or client) should acknowledge the sender, with ACK packets, just after a correct report packet is received and parsed. The receiver is responsible for parsing all the data contained within the REPORT packets.
3. The sender, as soon as an ACK is received, should stop sending report packets. This shall also happen when the maximum number of retransmissions is reached. In this case the connection should be considered lost and countermeasures should be taken by the application.
4. A new data exchange, if required, can then take place following the rules defined in (1), (2) and (3).

Note: the insertion of a timestamp inside ACK packets is not requires and it should not be in all the standard cases. If, however, a timestamp is added in order to perform additional specific operations in addition to the data exchange, it is up to the receiver of the ACK packet to manage such data.

XII. The client and server should then declare the LaMP session as terminated and continue (or terminate) their execution with other operations (including, possibly, starting a new LaMP session or reporting to the user all the gathered statistics about the measured latency values). A way to report at least the latency data should always be implemented in at least one entity (i.e. in at least one client or server for each session), following section 3.6.

3.5. Unidirectional operating mode

Other than the normal ping-like (bidirectional) operating mode, an additional unidirectional mode is supported by LaMP, to enable **one-way** latency measurements.

In this mode, the client should not expect any reply from the server, as the communication is always performed in one direction only (i.e. from client to server).

Each packet should always embed a timestamp, placed by the client, that shall be used by the server to try to compute the latency or perform other time-related computations, by extracting and comparing it to a local timestamp. As the two timestamps are now obtained inside different devices, this mode can work only when the clocks in the tested devices are very precisely synchronized, possibly with under-millisecond precision.

Since the way in which the clock are synchronized is not managed by LaMP, this mode should, at the moment, be considered as *experimental*.

LaMP, as of revision 2.0, shall rely on external protocols to keep the clocks synchronized (for instance the *Network Time Protocol* - NTP - or the *Precision Time Protocol* - PTP).

The operating mode should work as follows:

I. The connection is always initiated by the client, which should send an INIT packet to the server. The INIT packet may contain a timestamp in order to perform any time initialization operation from the server side, but it is not required and in the most common cases can be omitted (i.e. leaving it to 0, for what concerns both the **Sec** and **uSec** fields).
The initialization packet shall also contain a random ID, stored inside the 16 bit field of the LaMP header (it can be any number from 0 to 65535) and kept as it is until the session is terminated.
The "**Length or packet type**" field should be set to *0x0002*.

II. The server, upon receiving the INIT packet, should use its content to set the ID and the session type (i.e. *unidirectional*). Then, the server should accept only packets containing that ID and using the mode specified inside the "**Length or packet type**" field of the INIT packet. The server can optionally gather statistics about the INIT retransmission attempts, using the sequence numbers, and perform other user defined time related operations, in case a timestamp was inserted inside the packet.

III. After properly processing the INIT packet, the server should send back an ACK packet to the client, which will be informed about the server being ready to start the current measurement session.

IV. In the meantime, the client should repeat the transmission of the INIT packet, using a transmission interval defined by the user, until the ACK is received from the server or until a certain number of attempts is reached. In the last case, the client should terminate the current session. If no packets are lost, a single INIT-ACK transmission should occur, i.e. in ideal channel/link conditions, with no losses, the server should be fast enough to reply to the client before the first retransmission occurs and the client should not set a period which is too short, causing unnecessary retransmissions towards the server.

V. After the INIT-ACK procedure is completed, the client can start sending UNIDIR_CONTINUE packets. These packets must always embed a timestamp (that is also why there are no undirectional timestampless packets).
The timestamp can be obtained and placed inside the header at any time, using system calls such as *gettimeofday()*, under Linux, depending on the kind of latency measurement the user wants to perform.

Typically, the timestamp should be set in the last possible instant before sending the packet, in order to reduce the application latency as much as possible, but as this mode is *experimental*, it is up to the user to decide (and then document) when to set the timestamp.

The client should prepare all the unidirectional packets according to the rules specified before. It will set an initial sequence number, which will be then increased by 1 (or using a user defined rule) for each request transmission and optionally fill the payload.

If a payload is used, the "**Length or packet type**" field should be different than 0 and equal to the number of bytes contained inside the payload. The payload can contain random data, which will not be parsed by the server, in order to perform latency measurements with different payload sizes, or it can contain meaningful data which can be parsed by the by the server. This allows user defined per-packet data to be easily transmitted from the client to the server.

VI. The server should receive all the UNIDIR_CONTINUE and UNIDIR_STOP packets with the correct ID, but it shall not reply to the client. Instead, it shall extract the timestamp embedded inside the packet and compare it to a local receive timestamp in order to try to obtain latency measurement data. Please note that this mode, when the clocks are not very precisely synchronized, can provide meaningless data or even negative latency values.

In case negative values are detected, it is up to the user to decide what countermeasures to take.

VII. When the client is about to terminate its session, it should send a UNIDIR_STOP packet, respecting all the rules defined before in (V).

This will inform the server about the intention to terminate the current session, letting it perform memory cleanup, reporting, latency computation and other operations.

VIII. Upon reception of a UNIDIR_STOP packet, the server can then consider the session as terminated and can initiate the execution of the final memory cleanup and reporting operations. As the server is now responsible of reporting the latency information to the user or to other software modules, it should manage a *report* containing at least the fields defined in section 3.6.

The *report* can then be shown to the user, stored internally, sent to other running processes or sent to the client (see point (IX)), which can perform all these operation on the sender device.

IX. Optionally, the server can send back the report data to the client. The data exchange should follow the procedure described in point (XI) of section 3.4 (ping-like operating mode).

X. The client and server should then declare the LaMP session as terminated and continue (or terminate) their execution with other operations (including, possibly, starting a new LaMP session or reporting to the user all the gathered statistics about the measured latency values). A way to report at least the latency data should always be implemented in at least one entity (i.e. in at least one client or server for each session), following section 3.6; if no data exchange occurs, it shall be implemented inside the server.

3.6. Follow-up mechanism

Follow-up packets are types of packets which are sent just after normal LaMP packets, in order to provide additional time information which could not be embedded in the original packet.

Both the modes described in sections 3.4 and 3.5 support an additional follow-up mechanism, which can be used to provide a more precise estimate of the network latency or RTT, relying on follow-up packets which are sent after each reply (by the server in ping-like mode) or unidirectional packet (by the client in unidirectional mode) and which contain additional timestamp data to improve the initial measurement, which is normally based only on the time of transmission and reception of the request and reply packets.

The usage of this mode is optional, and it should be never set as the default one.

The follow-up mechanism is an addition to what is presented in sections 3.4 and 3.5 and it foresees the following additional steps:

I. After the INIT procedure, if the client wants to activate the follow-up mechanism, it should send a FOLLOWUP_CTRL packet, with its "**Length or packet type**" field set to *0x00__* (i.e. it should send a follow-up control **request** packet). The last byte should be set according the table presented in section 3.6.1.
The same session ID as the INIT procedure shall be used.

II. Upon the reception of the request, the server should reply with another FOLLOWUP_CTRL packet, using the same session ID, and with its "**Length or packet type**" field set to:
    - *0x0100* in order to **deny** the request, telling the client that no follow-up mode should be used. This may be the case for instance of an unsupported request type (see section 3.6.1) or an unsupported follow-up mode.
      In this case the client must not use the follow-up mechanism and the measurement session should start as described in sections 3.4 or 3.5 without involving any follow-up packet.
    - *0x0200* in order to **accept** the request, including the conditions imposed by the request type.
    - *0xFFFF* in order to inform the client about a **bad** or **unknown request**. As request types may not be implemented by the server, *0xFFFF* should be used to inform the client about an unknown, badly formatted or unimplemented follow-up type. This response should be treated as a deny and no follow-up mechanism should be used.

III. If the request has been accepted, the follow-up mechanism should be used until the end of the session.
In **ping-like mode**, the server, upon receiving each request and sending the corresponding reply, should estimate in the best possible way the time between the reception of the request and the transmission of the reply, i.e. its overall "processing time".
It should then prepare a FOLLOWUP_DATA packet, with an optional payload (which is normally not inserted), and embedding, in the timestamp fields of the LaMP header, the previously computed time difference. This packet shall be sent back to the client as soon as possible and just after sending the reply packet. The client should then report the measured RTT value by subtracting the server processing time, gathered from the follow-up packet, from its computed latency value (the one which could be computed without using the follow-up mechanism). Thanks to this additional timestamp data, the client entity shall be then able to obtain a more precise measurement of the network RTT, without taking into account the server processing time.
In **unidirectional mode**, the client should send to the server, after and in addition to each UNIDIR_CONTINUE or UNIDIR_STOP packet, a FOLLOWUP_DATA packet, embedding, in the timestamp fields of the LaMP header, the best possible estimate of the transmission timestamp, according to the request type negotiated before.

When the clocks are synchronized, this timestamp can be used to estimate in a more precise manner the client transmission time.

Follow-up packets in unidirectional mode are useful if and only if it is possible to obtain a more precise estimate of send timestamps than what can be included inside any UNIDIR_CONTINUE and UNIDIR_STOP packet, according to section 3.5.

3.6.1. Follow-up control request types

When sending a follow-up request, the "**Length or packet type**" field should be set with the first byte to *0x00* and the last byte specifying a type of request, as detailed in the following table.

It is not mandatory for a server to implement all the request types and follow-up mechanisms described below.

If LaMP is not implemented over an underlying operating system, only the follow-up modes resulting from the *0x00* and *0x03* request types can be implemented, together with other user defined requests, if needed. *0x01* and *0x02* shall always result in a *0xFFFF* reply, in this case.

Last byte	**Full "Length or packet type" field**	**Description**
0x00	0x0000	Use application level timestamps, when possible, to compute the timestamp data to be sent in the follow-up data packets. Timestamps should be gathered by the application itself trying to estimate the processing time/transmission time in the best way possible.
0x01	0x0001	Use kernel level receive timestamps, when possible, to compute the timestamp data to be sent in the follow-up data packets. Transmit timestamps are still at an application level.
0x02	0x0002	Use kernel level receive and transmit timestamps, when possible, to compute the timestamp data to be sent in the follow-up data packets.
0x03	0x0003	Use hardware level timestamps, when possible, to compute the timestamp data to be sent in the follow-up data packets.
0x04-0xEF	0x0004-0x00EF	Reserved for future use.
0xF0-0xFF	0x00F0-0x00FF	User defined request.

The meaning of the request type varies depending on the operating mode:

- In ping-like operating mode, it represents a requested type of follow-up timestamp data, which is requested by the client and accepted or denied by the server, which will, eventually, gather timestamps according to the kind of received request.
- In unidirectional mode, it represents a way to inform the server about which kind of timestamp data will be embedded by the client in the follow-up packets, which will be sent after each UNIDIR_CONTINUE or UNIDIR_STOP packet. Types *0x01* and *0x02* are the same as far as a unidirectional session is started.

3.7. Latency statistics standard format and required *report fields*

LaMP also defines a minimum number of data that should be gathered inside a *report*, when using it to measure the latency between different devices, and then used to inform the user or other applications/modules about the measured values.

This minimum number of data can be integrated with any number of user defined measurements and it is up to the application to manage it in a correct and efficient way.

It is important to highlight that each session should be performed by sending more than one request/reply, in order to obtain more meaningful data and filter out possible outliners.

The following table resumes the minimum set of measurements (each measurement is called *report field*) which should be performed for each session:

Required report field	Unit/Resolution	Description
Average latency	µs	Average latency over N measurements, as: $\frac{\sum_N L}{N}$ Where L is a single latency measurements, obtained thanks to the stored/sent timestamps.
Minimum latency	µs	Minimum latency value over N collected values.
Maximum latency	µs	Maximum latency value over N collected values.
Number of lost packets	#	Number of packets which were lost during the test. They shall be computed using the LaMP header sequence numbers.
Number of packets with errors	#	Number of packets which were not lost during the test, but which contained or caused errors, making the latency or RTT computation impossible. These packets shall be counted, but they shall never be used for the final latency statistics.

Each application should also document (and, optionally, report to the user) how the timestamps used to compute the latency values are obtained, possibly supporting more than one way of obtaining them. In this case, the type of latency measurement should be reported to the user in addition to the required *report fields* presented before.

3.8. Timeouts

In case some critical packets are lost, in particular PINGLIKE_ENDREQ, PINGLIKE_ENDREPLY (and their timestampless variants), UNIDIR_STOP, ACK and FOLLOWUP_CTRL, the application may risk to wait indefinitely for a packet that, potentially, may never be received, not terminating the current session as expected.

In this case, the application must set a receive timeout on any structure defined to receive LaMP packets. For instance, in case sockets are used, the application shall set a socket receive timeout.

It is completely up to the user to define a reasonable value.

When the timeout expires, the interested entity (a client or a server) shall perform all the memory cleanup and final operations to safely terminate the current session.

4. C library and API documentation

4.1. Rawsock library and Linux support

The LaMP protocol can be managed through the additional LaMP module of the **Rawsock** library (version 0.2.0 and newer).

As this library is Linux only, users can also define they own libraries to support LaMP, following the core API documentation in section 4.2.

As the **Rawsock** library is still at an initial development stage, any contribution or improvement, both to the library and/or this document is highly welcome.

The **rawsock_lamp** module, inside the **Rawsock** library, contains functions to populate the LaMP header, fill some specific fields and encapsulate an (optional) payload inside the LaMP packet, without making the user worry about structure packing, byte ordering or other low level network related problems.

This module also provides a structure, **struct lamphdr**, which contains all the header data and which can be used like other Linux structures for packet headers, such as *struct iphdr* or *struct ether_header*.

Each application using LaMP should declare a **struct lamphdr** and manage it though the available library functions. In particular, when sending a LaMP packet, the application should first populate the header with the required data, then insert an optional payload and merge it with the header, setting also the "**Length or packet type**" field, and send the packet using any underlying protocol.

When receiving LaMP packet, instead, the application must first fill a buffer with the complete packet, then extract the relevant information from the header and use them to check whether the packet is of interest. If it is (i.e. if it has the right type and ID), it shall proceed to extract from the header, using the *lampHeadGetData()* function, all the relevant information, including the timestamp (if the packet is not an ACK, INIT, REPORT or TLESS one) and use them to perform the operations described in chapter 3.

The complete library documentation can be found when downloading it and it is maintained and generated through *Doxygen*.

4.2. API documentation

In order to code your own library supporting LaMP, there is a minimal set of types, structures and functions that shall be defined.

To improve efficiency, some functions are defined to change specific header fields, in order to call the full header population function only once or, at least, the minimum possible number of times.

The user is then free to define any other useful function, type, macro or definition, which should be considered as a custom one and properly documented.

The required objects are listed in the following sections.

4.2.1. Required types

Type	Description
`byte_t`	8 bit unsigned integer to store a single byte. In C, this can be defined as `unsigned char.`
`lamptype_t`	Enumerator containing the LaMP packet types, strictly following the order of the table presented in section 3.1

`lamp_uint8_t`	8 bit unsigned integer. Can be replaced with `uint8_t`, if available (i.e. no need to define this type).
`lamp_uint16_t`	16 bit unsigned integer. Can be replaced with `uint16_t`, if available (i.e. no need to define this type).
`lamp_uint64_t`	64 bit unsigned integer. Can be replaced with `uint64_t`, if available (i.e. no need to define this type).
`struct timefields`	This structure should contain two fields to store a seconds and microseconds timestamp. It can be replaced, when available, with `struct timeval`.
`lamp_size_t`	This type should contain the length of any LaMP buffer. It can be replaced, when available, with `size_t`.

4.2.2. Required structures

Structure	Description

`struct lamphdr`	Structure containing the LaMP header. It shall have the following fields:
	`struct lamphdr {`
	`    lamp_uint8_t reserved;`
	`    lamp_uint8_t ctrl;`
	`    lamp_uint16_t id;`
	`    lamp_uint16_t seq;`
	`    lamp_uint16_t len;`
	`    lamp_uint64_t sec;`
	`    lamp_uint64_t usec;`
	`};`

4.2.3. Required macros and defines

Macro	Description
`#define ETHERTYPE_LAMP 0x88B5`	As LaMP can be encapsulated inside any other packet, when it is encapsulated directly inside an Ethernet packet it should carry the Local Experimental Ethertype, which can be specified by means of ETHERTYPE_LAMP. If the *0x88B5* is already defined as an operating system constant, it is suggested to use that instead of *0x88B5* inside this declaration.
`#define MAX_LAMP_LEN (65535)`	Maximum payload size, in bytes, that can be transmitted or received inside a LaMP packet.
`#define LAMP_HDR_PAYLOAD_SIZE(size) sizeof(struct lamphdr)+size`	This macro returns the size of a LaMP packet with a payload of `size` bytes.
`#define LAMP_HDR_SIZE() sizeof(struct lamphdr)`	This macro returns the size of a LaMP header.
`#define TYPE_TO_CTRL(field) (field | 0xA0)`	This macro allows the user to convert a `lamptype_t` value (specified as `fields`, 4 bits) to the corresponding full control field (`ctrl`) value (*0xA* + 4 type bits).
`#define CTRL_TO_TYPE(field) (field & 0x0F)`	This macro allows the user to convert a full control field (`ctrl`) value (*0xA* + 4 type bits) to the corresponding `lamptype_t` value (4 bits).

4.2.4. Required functions

Function	Full prototype and description
`lampHeadPopulate()`	`void lampHeadPopulate(struct lamphdr *lampHeader, unsigned char ctrl, unsigned short id, unsigned short seq);` This function should be used to populate a LaMP header (`struct lamphdr`), which is passed by the user as a pointer. The user shall specify an already existing LaMP header structure to be filled in, the control field value, as *ctrl*, the identification value and the sequence number of the current packet. This function should initialize both the timestamp fields to 0. Then, a call to `lampHeadSetTimestamp()` will be needed to insert a timestamp (or, if defined, any call to additional custom functions). This is not required when using timestampless packets. The function shall manage the byte order (i.e. the conversion from host to network byte order).
`lampHeadSetTimestamp()`	`void lampHeadSetTimestamp(struct lamphdr *lampHeader, struct timeval *tStampPtr);` This function takes as input an already prepared LaMP header structure and fills its timestamp field either with the data passed through the `tStampPtr` pointer (which should point to a custom timestamp) or, when this argument is NULL, with the current time, which can be obtained according to any valid method in the target operating system.
`lampSetUnidirStop()`	`void lampSetUnidirStop(struct lamphdr *lampHeader);` This function takes as input an already prepared LaMP header structure and sets its type to UNIDIR_STOP. It should be called to change the packet type in the `ctrl` field when the client is about to send the last packet of the current session.
`lampSetPinglikeEndreq()`	`void lampSetPinglikeEndreq(struct lamphdr *lampHeader);` This function takes as input an already prepared LaMP header structure and sets its type to PINGLIKE_ENDREQ. It should be called to change the packet type in the `ctrl` field when the client is about to send the last packet of the current session.
`lampSetPinglikeEndreqTless()`	`void lampSetPinglikeEndreqTless(struct lamphdr *lampHeader);`

	This function takes as input an already prepared LaMP header structure and sets its type to PINGLIKE_ENDREQ_TLESS. It should be called to change the packet type in the `ctrl` field when the client is about to send the last packet of the current session.
`lampSetPinglikeEndreqAll()`	`void lampSetPinglikeEndreqAll(struct lamphdr *lampHeader);` This function is a combination of the previous two. This function takes as input an already prepared LaMP header structure and it checks the packet type. If it is PINGLIKE_REQ, PINGLIKE_ENDREQ will be set as new type. If it is PINGLIKE_REQ_TLESS, PINGLIKE_ENDREQ_TLESS will be set as new type. It is neither PINGLIKE_REQ nor PINGLIKE_REQ_TLESS, no operation should be performed. It should be called to change the packet type in the `ctrl` field when the client is about to send the last packet of the current session and it should be preferred over the two previous functions, when possible.
`lampHeadSetConnType()`	`void lampHeadSetConnType(struct lamphdr *initLampHeader, lamp_uint16_t mode_index);` This function takes as input an already prepared LaMP header structure for an INIT packet and fills its "**Length or packet type**" field with the specified INIT type (passed as `mode_index`). For consistency reasons, the function should modify the specified header (passed through a pointer to a `struct lamphdr`) only when the "**control**" field is set to INIT (*0xA8)* and the INIT type (i.e. the `mode_index`) is a valid one, so if it is equal to *0x0001* or *0x0002*.
`lampHeadSetFollowupCtrlType()`	`void lampHeadSetFollowupCtrlType(struct lamphdr *followupLampHeader, lamp_uint16_t followup_type);` This function takes as input an already prepared LaMP header structure for a follow-up control packet and fills its "**Length or packet type**" field with the specified follow-up control type (passed as `followup_type`). For consistency reasons, the function should modify the specified header (passed through a pointer to a `struct lamphdr`) only when the "**control**" field is set to FOLLOWUP_CTRL (*0xAD)* and the FOLLOWUP_CTRL type (i.e. the `followup_type`) is a valid one, as defined before in this document.
`lampHeadIncreaseSeq()`	`void lampHeadIncreaseSeq(struct lamphdr *inpacket_headerptr);` This function takes as input the pointer to a LaMP header and shall be called to increase by one the sequence number field. Any other user-specific increasing scheme should be

	managed with custom functions; however, as the increase by 1 is mandatory for ACK, INIT and REPORT packets, this function should be defined.
`lampHeadGetData()`	`void lampHeadGetData(byte_t *lampPacket, lamptype_t *type, unsigned short *id, unsigned short *seq, unsigned short *len, struct timefields *timestamp, byte_t *payload);` This function takes as input the pointer to a LaMP header, obtained from a received LaMP packet. It then extracts all the relevant data from the header, including a pointer to the payload and the timestamp. The user shall be able to specify a NULL pointer for the fields which are not of interest, in that case the function should skip that part of the header. Instead, when non-NULL pointers are passed, it should store the header data in the memory locations specified by the user. When a payload can be present, the function shall always check if the "**Length or packet type**" is 0. If it is, it should not store anything in the memory area pointed by `payload`. If it is not and the pointer is non-NULL, it should copy the LaMP payload content inside the memory area pointed by `payload`. It should also manage the byte order (i.e. the conversion from network to host byte order).
`lampEncapsulate()`	`void lampEncapsulate(byte_t *packet, struct lamphdr *lampHeader, byte_t *data, lamp_size_t payloadsize);` This function combines a LaMP (optional) payload and header, the latter in the form of a `struct lamphdr` structure. It can be used if an additional payload needs to be inserted inside a LaMP packet. If the application does not need to insert any payload, it can directly send the LaMP header structure as payload of any other protocol, encapsulating LaMP, without the need of calling `lampEncapsulate()`. The function takes as input a buffer in which the full packet will be put (it should be already allocated to the correct size), the LaMP header, the pointer to the buffer containing the payload (`byte_t *data`) and its size in bytes. This function must set the "**Length or packet type**" field, depending on the length of the payload which is combined with the header and properly manage its byte order (i.e. the conversion from host to network byte order).

`lampGetPacketPointers()`	`byte_t *lampGetPacketPointers(byte_t *pktbuf, struct lamphdr **lampHeader);` This function should be used when a LaMP packet with an additional payload is received. Given the full LaMP packet buffer (`pktbuf`), as an array of bytes, the function should fill a LaMP header structure pointer with the pointer to the LaMP header inside `pktbuf`, returning the pointer to the payload section inside `pktbuf`. No memory is allocated by this function: it only performs pointer arithmetic to return the proper pointers inside a full packet buffer. The case in which a packet with no payload is passed to this function should be managed and the function should return NULL if no payload exists.