The MCL Multicast Library: Concepts, Architecture and Use*

Vincent Roca

INRIA Rhône-Alpes, projet Planète, Grenoble, FRANCE
vincent.roca@inrialpes.fr - http://www.inrialpes.fr/planete/people/roca

Work in Progress, Describes MCL version 2.0 - October 4th, 2000

Abstract

This paper describes the rationals, the design and the use of the MCL MultiCast Library. This library is in particular ALC (Asynchronous Layered Coding) compliant and provides an efficient solution for highly scalable reliable multicast transmissions.

Contents

*This work has been partially done with LIP6-CNRS resources till September 2000, and with INRIA-R.A. resources (since October 1999). Part of this work has also been done in the context of the Distributed Systems Engineering (DSE) European Community project IST-1999-10302.

1 Introduction and goals of the MCL library

This paper addresses the multicast distribution of data flows generated on the fly to a set of widely heterogeneous receivers, a common situation with for instance collaborative work applications. It discusses the design of a *library for multicast transmissions* - MCL, MultiCast Library - that aims at being easily used in legacy or new applications.

Starting from version 2.0, this library *complies with the IETF standard for multilayer multicast transmissions, ALC (Asynchronous Layered Coding)* [2]. ALC is a recent massively scalable reliable multicast protocol proposed by the RMT working group that relies on multiple multicast groups. Using multiple multicast groups enables each receiver to receive data at an appropriate rate by subscribing to as many groups as made possible by their networking (or processing power) capabilities. ALC builds upon many previous works in the area of multi-layer data organization schemes [10][7], congestion control for multi-layer transmission schemes [4][11], and forward error correction (FEC) [6][3].

ALC can be used by many different applications for either “on-demand”, “streaming” or “push” delivery of data. In “on-demand” delivery, data is sent continuously and receivers arrive at any time. This mode is well suited to the distribution of popular files (e.g. a video-clip, the latest Linux distribution, etc.). In “streaming mode” a receiver typically remains joined for a long period, receiving many objects (e.g. images) produced in real-time. In “push” mode, all the receivers must be ready before the transmission starts. This mode is well suited to the plannified distribution of files (e.g. a database between the headquarters and various agencies).

2 General architecture

In this section we outline the library architecture and its consequences, and define the services it provides.

2.1 Definition of the MCL service

The service provided to the application is the *multicast delivery* of messages (called ADUs, or Application Data Units, in this paper) to a set of unknown receivers. These receivers can be *heterogeneous* (in terms of networking possibilities, host performances, or user desires) and one goal is to offer the possibility for *everybody* to receive data at an appropriate speed (i.e. as close as possible to the receiver’s fair share).

This delivery *preserves ADU boundaries* (as with UDP). If segmentation is required to comply with PMTU constraints, then ADUs will be reassembled before being delivered to the receiving application. We try to avoid as much as possible IP fragmentation which is considered inefficient [5]. Therefore each segment has a “small” size (512 bytes) to guaranty that it will not be further fragmented within the network.

Even if we use the term of ADU, the service provided is not necessarily ALF compliant [1]. Depending on the application, a receiver can ask either to receive ADUs in sequence, which may delay their delivery, or in arrival order.

The two delivery services mentioned in section 1, *on-demand* and *push*, are provided. Depending on the application needs, transmissions can thus either be cyclic to enable any potential receiver to arrive at its discretion, receive the ADUs and leave, or at a given predefined time for sender-oriented synchronous file distributions.

The provided service is *not necessarily reliable*. Reliability is regarded as an optional property that is enabled, when necessary, by the application. Reliability can be total, which requires either a feedback ACK/NAK mechanism (not available in release 2.0), or a cyclic transmission scheme (losses in a cycle can be recovered during the next cycle). Reliability can also be partial, meaning that some ADUs may not be delivered to the receiving application. In that case the ADU delivery must of course be in “arrival order”.

2.2 MCL internal building blocks

Figure 1 illustrates the general architecture. It shows:

- the library implemented in user-space,
- its internal functional blocks, namely:

  - congestion control for TCP friendliness on WAN communications (version 2.0 supports RLC [11]),
  - (a certain level of) reliability thanks to the use of a FEC codec [3] (version 2.0 supports a Reed-Solomon codec [6]),
  - data scheduling over the various layers (version 2.0 uses a random permutation of the various packets as suggested in [2]),
  - the optional ODL (On-Demand Layer Addition) [8] building block,
  - the internal library threads, etc.

- its upper Application Programming Interface (API),
- the use of the Socket library and UDP/multicast IP stack.

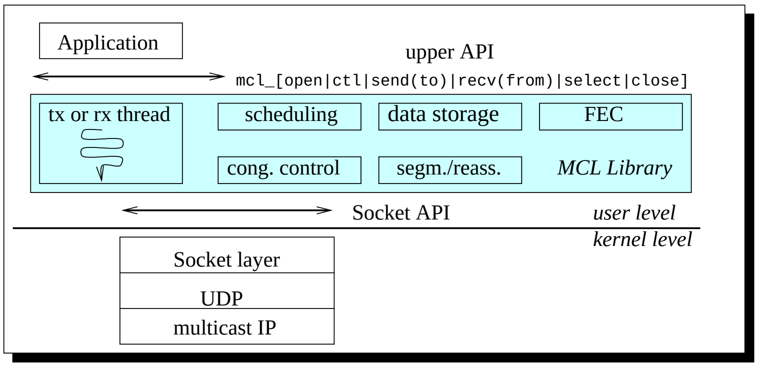

Figure 1: General architecture of the MCL library.

The upper API has been kept as simple as possible in order to facilitate its integration in existing applications. It is composed of only six functions (see section 3.1):

- `mcl_open()`
- `mcl_ctl()`
- `mcl_send()`
- `mcl_sendto()`
- `mcl_recv()`
- `mcl_recvfrom()`
- `mcl_select()`
- `mcl_close()`

The default behavior of the library is essentially controlled by constant definitions gathered in a dedicated file (see section 3). It is therefore possible to adapt the transmission profile to a given application profile at compilation time. If some parameters must be modified in real time or at application start, then the `mcl_ctl()` function can also be used.

2.3 Consequences of an application-level implementation

Being implemented at application level, it has several limitations:

- extra data copies to/from the application buffers are required within the library to avoid modifying the application buffering scheme[1].

[1] An optimization that could save the need for extra copies consists in unifying the application and library buffering: the application agrees to completely delegate buffer management to the library after a mcl_send(). This is not possible in this version because we want to make the application migration to MCL as simple as possible. More generally, MCL aims at optimizing network communications rather than CPU processing.

- `mcl_close()` needs to be synchronous (i.e. blocking) in order to enable the library to finish transmissions (it can take quite a long time on layer 0)[2]
- `fork()` and `exec()` system calls are not possible *during the use* of MCL services since they would respectively duplicate and erase the MCL context. Yet they can still be used before the opening of the MCL service.
- The handling of the `select()` system call must be performed by the MCL library. Indeed the identifier returned by the `mcl_open()` function does not designate any socket and therefore cannot be used by the usual `select()` system call.

3 Using the MCL library

3.1 Application Programming Interface (API) definition

The following functions define the user interface to the MCL control layer and are defined in the `MCL_ROOT_DIR/src/mcl_lib.h` file (to include in your source files). It has been kept as simple as possible in order to facilitate the migration process.

mcl_open: Open a MCL access point and perform necessary initialization. This function must be called first. It returns a session identifier.

This call is blocking (synchronous).

```
int      mcl_open        (char *mode);
```

mcl_ctl: This function is used to customize the MCL library: change the default multicast address, default port number, change the base layer transmission rate, etc. It provides a single, easily extensible, interface to MCL configuration that looks like the `setsockopt()` Socket system call. The following options are currently available:

```
int      mcl_ctl         (int id, int optname, void *optvalue, int optlen);

/*
 * library options set with the mcl_ctrl() function
 */
enum mcl_opt {
        /*
         * common options used by both sender and receiver
         */
        MCL_OPT_LEVEL = 1,      /* nb of tx levels (or layers) */
                                /* argument: int */
        MCL_OPT_PORT,           /* port number (uni or multicast), HOST order*/
                                /* argument: int (and not ushort!) */
        MCL_OPT_ADDR,           /* address (uni or multicast), HOST order */
                                /* argument: int */
        MCL_OPT_BIND,           /* specify sockaddr struct, NETWORK byte order*/
                                /* argument: struct sockaddr_in */
        MCL_OPT_TTL,            /* specify TTL (time to live), ushort expected*/
                                /* argument: int */
        MCL_OPT_STATS,          /* collect various stats */
                                /* argument: int */
        MCL_OPT_VERBOSITY,      /* verbosity level */
                                /* argument: int */
        MCL_OPT_DEBUG,          /* no tx/rx thread to simplify debug */
                                /* argument: none */
        MCL_OPT_MOREABOUT,      /* version # and credits */
                                /* argument: none */
        MCL_OPT_DELIVERY_MODE,  /* specify the delivery mode: push/on-demand */
                                /* argument: int */
```

[2]This is the same behavior as that of the close() Socket system call with TCP in SO_LINGER mode [9].

```
        /*
         * options used only by the sender
         */
        MCL_OPT_DATAGRAM_SIZE,  /* datagram size */
                                /* argument: int */
        MCL_OPT_TX_RATE,        /* base tx rate in full-sized packets/s */
                                /* argument: int */
        /*
         * options used only by the receiver
         */
        CL_OPT_IMMEDIATE_DELIVERY, /* boolean specifying if ADU can be */
                                /* delivered in arrival order or not */
                                /* argument: int (0 or 1) */
        MCL_OPT_RECVALL,        /* receive all DUs (for tests only!) */
                                /* argument: none */
        MCL_OPT_SEQNB_STATS,    /* collects stats for raw sequence # plotting */
                                /* argument: none */
        /*
         * RLC (congestion control) options (no need to change in theory)
         */
        RLC_OPT_SP_CYCLE,       /* Interval between 2 SPs at layer 0 (in sec)*/
                                /* argument: int */
        RLC_OPT_PKT_TIMEOUT,    /* Time To Wait for a late packet to arrive */
                                /* before assuming it's lost */
                                /* argument: int */
        RLC_OPT_DEAF_PERIOD,    /* Deaf period after dropping a layer */
                                /* argument: int */
        RLC_OPT_LATE_ACCEPTED,  /* number of late packets accepted between */
                                /* 2 SPs at highest layer */
                                /* argument: int */
        RLC_OPT_LOSS_ACCEPTED,  /* number of lost packets accepted between */
                                /* 2 SPs at highest layer */
                                /* argument: int */
        RLC_OPT_LOSS_LIMIT,     /* loss limit before layer drop */
                                /* argument: int */
        RLC_OPT_LOSS_TIMEOUT,   /* timeout for loss limit */
                                /* argument: int */
};

/*
 * delivery mode (cf ALC i-draft)
 */
#define DEL_MODE_PUSH           0       /* tx once, assumes synchronized rx */
#define DEL_MODE_ON_DEMAND      1       /* tx cyclically, rx start at any time*/
```

This call is non blocking.

mcl_send: Send an ADU (i.e. application buffer).

```
int     mcl_send        (int id, char *buf, int len);
```

This call is non blocking (asynchronous).

mcl_sendto: Like `mcl_send` but the application specifies the destination address (unicast).

```
int     mcl_sendto      (int id, char *data, int len,
                         struct sockaddr *saddr, int saddr_len);
```

This call is non blocking (asynchronous).

mcl_recv: The application asks for an additional ADU. The buffer should be large enough to contain the whole ADU, otherwise data is truncated and an error message returned.

```
int      mcl_recv        (int id, void *buf, int len);
```

This call is blocking (synchronous).

mcl_recvfrom: Like `mcl_recv` but the application is informed of the sender's address. This is important if an answer is required.

```
int      mcl_recvfrom    (int id, void * buf, int len,
                          struct sockaddr *saddr, int *saddr_len);
```

This call is blocking (synchronous).

mcl_select: Specify a maximum waiting time while listening to read/write/exception events. Works with both standard file descriptors (e.g. a socket) and MCL identifiers *at the same time*.

```
int      mcl_select      (int n, fd_set *readfds, fd_set *writefds,
                          fd_set *exceptfds, struct timeval *timeout);
```

This call is blocking (synchronous).

mcl_close: The application has no longer any data to send (sending side) or to receive (receiving side). Do whatever is necessary and close the MCL access point.

```
int      mcl_close       (int id);
```

This call is blocking (see section 2.3) (synchronous) on the sending side.

3.2 Installation of the library and the associated tools

Here are the necessary steps to build the library and the associated tools:

- edit the `MCL_ROOT_DIR/src/mcl_profile.h` and chose the profile you need (this is the only file you may need to modify). The idea is to define at compilation time a default behavior and to call the `mcl_ctl()` function at runtime for dynamic modifications only.
- have a look at `MCL_ROOT_DIR/Rules.mk` to check the target OS. You may also change some compilation flags in each `MCL_ROOT_DIR/*/Makefile` (e.g. compile in `DEBUG` mode or not, use flag `-g` or not, etc.)
- compile everything with `gmake` at the top directory `MCL_ROOT_DIR/`. Using `gmake` is required by the use of some makefile extensions. In some cases, `make` is an alias to `gmake` (e.g. with Linux).

In addition to the `MCL_ROOT_DIR/src/libmcl.a` library, the following tools will be compiled:

bench: This is a traffic generator for performance measurements. It is a modified version for MCL of the bench tool available at URL: `http://www.inrialpes.fr/planete/people/roca/bench/`

mclftp: This is a tiny file transfer protocol. Its goal is to prove that using MCL is really, really simple!

fcast: This is an implementation of the multicast file transfer application defined in [2]. For increased reliability, an application level checksum is appended by the sending side and checked by each receiver. The file name (and possibly other information) is sent along with the data in a standardized meta-data trailer.

If you want to port a legacy application on top of MCL, you need to:

- identify in your application all the Socket (or XTI/TLI) system calls and replace them with `mcl_XXX` function calls.
- compile your application with: `-LMCL_ROOT_DIR/src/ -lmcl`. See examples in `MCL_ROOT_DIR/fcast/Makefile`
- that's all (at least in theory `;-)`

4 Detailed architecture

We now go into more details and describe some MCL internal aspects.

4.1 The various data unit types

Three kinds of data units are used:

ADU: Application Data Units are the data messages submitted by the application. They are segmented to comply with PMTU limitations (a default 512 bytes length is used since multicast PMTU discovery is not feasible) and they are reassembled by the receiving MCL instance in order to preserve the application framing semantics.

```
typedef struct adu {
        struct adu      *prev, *next;
        int             seq;            /* seq nb of this ADU */
        int             len;            /* number of bytes in this ADU */
        struct block    *block_head;    /* first block of this ADU in list */
        int             block_nb;       /* number of blocks in this ADU */
        struct sockaddr saddr;          /* ADU recved from or destined to */
        int             saddr_len;      /* length of addr */
        /* fields only used by the sender */
        char            *data;          /* ptr to data buffer */
        /* fields only used by the receivers */
        char            ia_adu;         /* boolean: 1 if implicitly announced*/
        char            completed;      /* boolean: 1 if recvd all the blocks */
} adu_t;
```

Block: Blocks are the result of the segmentation of ADUs into several blocks of size appropriate to small block FEC codecs.

```
typedef struct block {
        struct adu      *adu;           /* ADU to which this block belongs */
        int             seq;            /* block sequence number */
        int             len;            /* number of bytes in this block */
        struct du       *du_head;       /* first DU of this ADU in list */
        int             du_nb;          /* nb of DUs that compose this block */
#ifdef FEC
        struct du       *fec_du_head;   /* first FEC DU of this ADU in list */
        int             fec_du_nb;      /* nb of FEC DUs in this block */
#endif
        /* fields only used by the receivers */
        int             du_rx;          /* nb of non-FEC DU recvd (!= du_nb) */
        char            completed;      /* boolean: 1 if recvd all the DUs */
} block_t;
```

DU: Data Units are the result of the segmentation of the block or the result of FEC packet creations. They are actually the data payload of the packets sent.

```
typedef struct du {
        struct du       *prev, *next;   /* links used only by the receiver! */
        struct block    *block;         /* block to which this DU belongs */
        int             seq;            /* sequence number of this DU */
        int             len;            /* number of bytes in this DU */
        char            *data;          /* ptr to data in buffer */
#ifdef FEC
        char            is_fec;         /* 1 if FEC du, 0 if plain data DU */
#endif
} du_t;
```

Table 1 compares the official ALC terminology with ours.

At the sender, the MCL library maintains a (doubly linked) list of ADUs (in the txlvl_tab[0] structure). For each ADU entry, the sender creates a table of block_t structures. Each block_t contains in turn two tables: a source DU table, and a FEC DU table created by the FEC codec.

Table 1: MCL/ALC Terminology and Numbering.

MCL terminology	ALC terminology	Identified by
ADU	Object	ADU sequence number (starts at 1)
block	block	block sequence number (starts at 0)
source/FEC DU	source/FEC symbol	DU seq number (starts at 0) plus is_fec flag (ie. source DU: $[0; max]/0$; fec DU: $[0; max]/1$)

At the receiver, the MCL library maintains a (doubly linked) list of ADUs announced, a table of their constituting blocks, and a (doubly linked) list of the DUs received for each ADU/block (in the rxlvl_t structure).

4.2 ALC packet header

Figure 2 shows the packet header used by MCL. It complies with the ALC draft version 01 (July 2000) recommendations while doing some choices.

SP	reserved	layer	sequence id	
0	hdr_len	fec id (1)	1 E 0	0 TLL TIL 0 0 X
payload id: block seq id (24b) | du seq id (8b)				
object tx label: ADU id (32b)				
object tx info: ADU len (32b) block len (32b)				

Figure 2: RLC/ALC packet header.

4.2.1 RLC header

The RLC (Receiver Driven Layered Multicast Congestion Control) header consists in a single 32 bit word. It includes a `SP` (synchronization point) flag to indicate when a receiver can add a layer, a `layer` field to indicate the target layer on which this packet is sent, and a `sequence id` field to identify packet losses (i.e. gaps in the `sequence id` of received packets). See [11] for more information on RLC.

4.2.2 ALC header

The ALC header immediately follows the RLC header. It is composed of:

ALC fixed-size header: The first 32 bit contains information (like the header length, not comprising the RLC/this 32 bits), several flags that determine the presence and actual size of the following fields.

- `TLL` (OTL Length) determines if the OTL field is present (this is always the case if the packet contains some data) and its length (in our case it should contain 1 to indicate that OTL is 32 bit long).
- `TIL` (FTI Length) determines if the FTI field is present (this is always the case if the packet contains some data) and its length (in our case it should contain 2 to indicate that FTI is 64 bit long).
- `X` (eXtension Header) indicates if one or more extension header(s) are present. In our case these extension headers are only used by `NO_NEW_ADU` and `CLOSE` signals (see below).

FEC payload ID (FPI): The goal of this field is to identify the content of the ALC payload. To that purpose, this field is composed of:

- the block sequence number (24 bits)
- the DU sequence number (8 bits)

Choosing a small DU sequence number field is due to the use of a small block FEC coded (Reed-Solomon) which limits in practice the parameters k to small values (typically a few tens).

Object transmission label (OTL): The goal of this field is to identify the object to which the current DU belongs. This field is composed of the ADU sequence number (32 bits). This field is required as MCL supports multi-object sessions.

FEC object transmission information (FTI): The goal of this field is to give some information on the object and its FEC coding. This field is composed of:

- the ADU length in bytes
- the block length in bytes (this is the length of a full-sized block, in practice the last one may be smaller)

In addition to this fixed-sized header and associated fields, MCL uses several extension headers to convey additional signaling information.

4.3 Signaling messages

The following signaling messages are defined:

NEW_ADU: A new ADU has been submitted by the application. This message is *implicitly conveyed* by the first packet of a new ADU and the related information is found in the FTI field. Therefore since version 2.0 MCL no longer defines any `NEW_ADU` message even if it still appears in the Finite State Machine.

NO_NEW_ADU: The sending application has informed MCL that it will no longer submit new ADUs (this happens when the `mcl_close()` function is called). This information is sent to the receiver(s).

CLOSE: The application has informed MCL that it will not submit any new ADU *and* MCL has currently sent all remaining packets. Therefore we enter this final status (close) and send the information to the receiver(s).

These messages are sent in dedicated signaling extension headers (see section 4.2).

To avoid problems related to the loss of signaling messages, *packets are made self-sufficient.* It means that each DU packet contains all the ADU information required to process it (e.g. create a new ADU context and store the DU in the appropriate block of this ADU). Similarly, once `NO_NEW_ADU` has been sent, all the following packets contain the same `NO_NEW_ADU` extension header. Thereby the loss of any packet does not compromise the good synchronization between the sender and the receivers. The `CLOSE` signal is sent several times (`MAX_TX_SIG_CLOSE`) in dedicated packets (since by definition all DUs have already been sent).

4.4 Transmission and reception control blocks

Information required to the transmission and reception sides are respectively stored in several control blocks.

- On the sending side there is one control block *per transmission level.* This is implemented as an array of the `txlvl_t` structure.
- On the receiving side there is only a *single*, much simpler, control block. This control block is a variable of `rxlvl_t` type.
- In addition there is a `mclcb_t` structure per session where all the features of that session are kept (`txlvl_t` and/or `rxlvl_t` structures, addresses, number of layers, statistics, etc.).

4.5 Goal of the main thread

MCL behavior is controlled by a thread that wakes up at a predefined frequency. Here is a simplified description of this thread:

```
while (1) {
        usleep(10000);  /* 10 ms */
        pthread_mutex_lock(&global_lock);
        time_count++;
        call the various function timers; /* (e.g. RLC, ODL, etc.) */
        /* see what to send next for each tx level */
        something= 0;
        for (i = 0, tl = txlvl_tab; i < max_level; i++, tl++) {
                if ((nb = min(tl->du_per_tick, tl->tot_rem)) > 0) {
                        /* something to send ! */
                        SendIt(tl, nb);
                        something = 1;  /* something has been sent */
                }
        }
        pthread_mutex_unlock(&global_lock);
        if (no_new_ADU && something == 0) {
                /* finished, stop everything */
                SendClose();
                exit (0);
        }
}
```

The thread is periodically awaken. It scans all the transmission levels to determine if data needs to be sent (i.e. if there remains unsent data). If yes, it calls the transmission function: `SendIt()`.

The `time_count` global variable determines the current virtual time.

In addition it calls the protocol timer function of the various building blocks (if any) for periodic processing. Examples are the RLC timer (to scan late packets and decide if they are considered as definitely lost or not) and the ODL timer.

4.6 Goal of the reception thread

A reception thread is also required. Its goal is to receive data as soon as available in the reception UDP Socket. Indeed, if we only rely on the application to retrieve data (i.e. to call `mcl_recv()` which in turns calls the `select()/recv()` system calls), there is a risk that the socket buffer overflows thereby creating additional losses. To avoid it, we use an additional thread that always waits for data to be available (`select()` / `recv()` system calls). Because the buffering capabilities at MCL level are important (being at application level it takes advantage of the virtual memory system), we avoid any loss, even in case of non greedy receiving application.

4.7 Example: sending an ADU

This example illustrates the processing performed from the moment the application submits a new ADU:

```
mcl_send(buf, len)
        Create an adu_t structure;
        Copy data to an internal buffer of the ADU;
        Insert the ADU in the txlvl_tab[0] list;
        Call SegmentADU(adu)
                Find the ADU sequence number;
                Segment the ADU, i.e. create an array of blocks
                Segment each block in multiple source DUs
        For each block, create a list of FEC DUs;
        Call UpdateTxPlanning(dub)
                Identify what data to send (source or FEC DU) on
                each transmission layer at any time;
                Store this sequence in an internal tx table;
        Return the control to the application;
```

Nothing is sent at that time for this new ADU. Transmissions are performed later by the transmission thread, at a regular rhythm as described in section 4.5.

4.8 On the use of sockets

Two kinds of sockets are used by the library:

session socket: this socket is used to receive data packets multicast'ed to the group. It is bound (`bind()` system call) to the public port number of the session (supposed well known) and `IP_ADD_MEMBERSHIP` has been used to receive the multicast traffic. Yet no transmission is ever performed from this socket!

private socket: this socket is used to send unicast and multicast traffic and to receive signaling only packets unicast'ed to us. Unlike the session socket, here we issue no `bind()` system call in order to get a private port number, locally unique.

We also distinguish between:

data traffic : packets containing a DU. Note that signaling can be included too (in-band signaling) (e.g. to carry a `NONEWADU` message).

signaling traffic packets sent in unicast usually and containing only signaling information (out-of-band signaling). This is used for instance by the ODL protocol and to carry a `CLOSE` message.

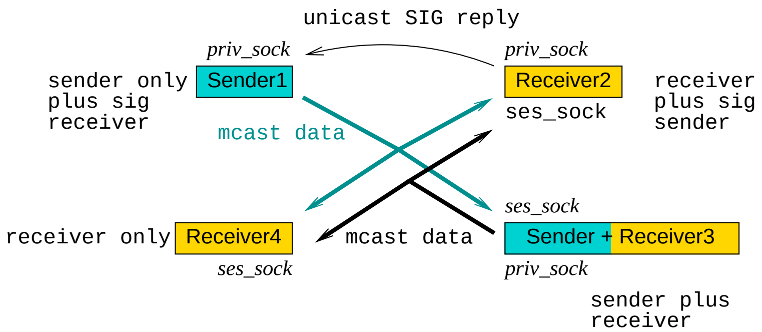

Figure 3: Kinds of traffic exchanged between hosts.

Figure 3 illustrates all the kinds of traffic that are used according to the type of host (pure sender, sender and receiver, pure receiver). Data traffic flows from sender(s) to receiver(s), while signaling traffic are either piggy-backed, or unicast'ed from a receiver back to a sender, or multicast'ed by a receiver to all group members.

4.9 Statistics gathering

Many statistics are gathered. Some of the fields of the `stats_t` structure are collected by the sender, others only collected by the receiver. Some statistics (end of ADU reception) are stored in a circular buffer to limit memory usage. See file `mcl_stats.h` for more information.

These statistics can be printed by either the sender or the receiver periodically or after each end of ADU reception. Here is an example of statistics gathered at the sending side:

```
--- TX STATISTICS ---
        time=51  tx_bytes=56835  tx_fec_bytes=64000 (data only)
        tx bytes per layer:     24167   24167   24167   24167   24167 0       0
time=51  tx_pkts=125  tx_fec_pkts=125
        tx pkts per layer:      50      50      50      50      50      0       0
        Data+SIG: tot_snt=127475  tot_rx=0 bytes
        buf_space=11367  max_buf_space=11367  adu_announced=0
```

. . . and at the receiving side during the same session:

```
--- RX STATISTICS ---
        time=73  rx_bytes=12434  rx_fec_bytes=13824  dup_bytes=14435  dup/rx=116.09  rx/undup=0.00
        bytes per layer:        23655    2603   0       0       0       0       0
        time=73  rx_pkts=28  rx_fec_pkts=27  dup_pkts=30  dup/rx=107.14  rx/undup=0.00
        pkts per layer:       49      6       0       0       0       0       0
        undup pkts per layer:   22      3       0       0       0       0       0
        Data+SIG: tot_snt=0  tot_rx=27970 bytes
        errors=0  adu_announced=3  adu_completed=3

--- END OF RX TIMES ---
                seq=3 time=73
                seq=2 time=71
                seq=1 time=44
```

4.10 The integrated application

An integrated application has been designed to perform tests. It consists of two parts:

- a client that submits ADUs according to a given scenario (periodic submissions or more elaborated scenario including several steps)
- a server that receives ADUs and performs some statistics.

This application is integrated in the library (rather than being built on top of the API) and can be found in the source directory. It is activated by the transmission thread to perfectly control the ADU submissions. All tests carried out with this application are thus *reproducible* (deterministic). Its use is controlled by a constant defined in file `mcl_profile.h`.

5 Availability

The MCL library and its associated tools are available at URL:
`http://www.inrialpes.fr/planete/people/roca/mcl/}`

To simplify, MCL is a *free of use tool as long as you do not try to make money with it* in any way and *is provided in source code format.* The official copyright text can be found in the distribution.

6 Credits

Several people participated in the MCL library design:

- *Vincent ROCA* (1998 up to September 2000, LIP6; since October 2000, INRIA-R.A.): head of the MCL project; design and implementation of most of the MCL's building blocks.
- *Julien LABOURE* (since May 2000, INRIA-R.A.): RLC, fcast, ALC headers
- *Christophe CAVROT* (May-August 1999, LIP6): FEC integration

The following research institutes participated in the project:

- LIP6 - CNRS, thème Réseaux et Performances, Paris, FRANCE
- INRIA Rhône-Alpes, projet Planète, Grenoble, FRANCE

Since mid-2000, this work is also partially done in the context of the *Distributed Systems Engineering (DSE)* European Community project IST-1999-10302.
The FEC module used by the MCL library is derived from:

```
fec.c -- forward error correction based on Vandermonde matrices
(C) 1997-98 Luigi Rizzo (luigi@iet.unipi.it) (980624)
Portions derived from code by Phil Karn (karn@ka9q.ampr.org),
Robert Morelos-Zaragoza (robert@spectra.eng.hawaii.edu) and
Hari Thirumoorthy (harit@spectra.eng.hawaii.edu), Aug 1995
```

References

[1] D. Clark and D. Tennenhouse. Architectural considerations for a new generation of protocols. In *IEEE SIGCOMM'90*, September 1990.

[2] M. Luby, J. Gemmell, L. Vicisano, L. Rizzo, J. Crowcroft, and B. Lueckenhoff. *Asynchronous Layered Coding (ALC): a massively scalable reliable multicast protocol*, July 2000. Work in Progress: <draft-ietf-rmt-pi-alc-01.txt>.

[3] M. Luby, J. Gemmell, L. Vicisano, L. Rizzo, J. Crowcroft, and B. Lueckenhoff. *Reliable multicast transport building block: Forward Error Correction codes*, July 2000. Work in Progress: <draft-ietf-rmt-bb-fec-01.txt>.

[4] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *ACM SIGCOMM'96*, October 1996.

[5] J. Mogul and S. Deering. *Path MTU discovery*, November 1990. RFC 1191.

[6] L. Rizzo and L. Vicisano. Effective erasure codes for reliable computer communication protocols. *ACM Computer Communication Review*, 27(2), April 1997.

[7] L. Rizzo and L. Vicisano. Reliable multicast data distribution protocol based on software fec techniques. In *Fourth IEEE Workshop on the Architecture and Implementation of High Performance Communcation Systems (HPCS'97), Greece*, June 1997.

[8] V. Roca. On the use of on-demand layer addition (odl) with multi-layer multicast transmission techniques. In *2nd Workshop on Networked Group Communication (NGC2000)*, November 2000.

[9] R. Stevens. *TCP/IP illustrated, volume 1: the protocols*. Addison-Wesley Publishing Company, 1994.

[10] L. Vicisano. Notes on a cumulative layered organisation of data packets across multiple streams with different rates. Research Note Note RN/98/25, University College London (UCL), May 1998.

[11] L. Vicisano, L. Rizzo, and J. Crowcroft. Tcp-like congestion control for layered multicast data transfer. In *IEEE INFOCOM'98*, February 1998.