Multicast Support in XEmacs

Didier Verna
The XEmacs Project
<didier@xemacs.org>

April 19, 1999

Abstract

This paper describes the recent multicast support that have been added to XEmacs, publicly available for the first time in XEmacs 21.0. First, a general overview of the "multicast" technology is given. Then, the internal support within XEmacs is described and the first multicast package for XEmacs is presented. The current problems and perspectives with this support are finally detailed.

Introduction

Aside from its incredible capabilities for text edition and extensibility, one of the greatest features of XEmacs is probably its capacity to communicate with the rest of the world: thanks to the process abstraction layer, you can browse the web, compile programs, run shells inside an XEmacs frame. Until XEmacs 21.0, only one type of network connection was supported: the traditional TCP streams. This paper presents the support for "multicast", a special type of network connection that was recently added to XEmacs. Section 1 gives a general overview of what multicast is and why it is useful. Section 2 describes how the multicast support is implemented in XEmacs, and section 3 presents MChat, the first multicast package written for XEmacs. As this feature is still experimental, the last section describes current problems that should be solved in order to get a satisfactory and complete support, and future work that is currently planned.

1 The Multicast Technology: a General Overview

1.1 Multicast vs. Unicast and Broadcast

Multicast is a special kind of network connection, based on the UDP protocol, which lets you establish a "group" connection rather than a "point"

connection. Figure 1 illustrates the conceptual difference between unicast and multicast connections.

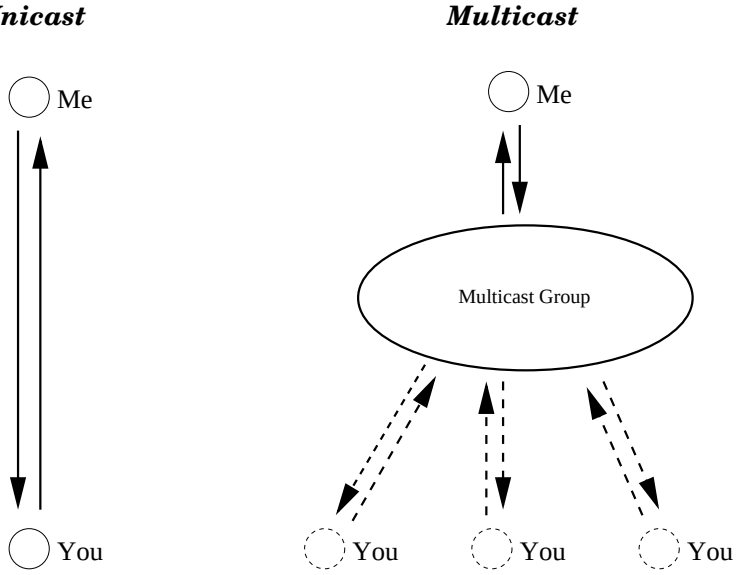

Figure 1: Unicast vs. Multicast

In a traditional unicast network connection, you want to contact another address (say, a client wants to contact a server) and establish a connection for writing and reading to and from this remote address. This philosophy is called "point-to-point". On top of this, we find "broadcast" connections. Broadcasting means that you send a message to several remote addresses at the same time. In that case, the connections are usually unidirectional, meaning that only you will write messages to the other addresses. Moreover, you actually need to know and connect to *all* the remote addresses, and send the message once for each remote address.

Multicast is a different philosophy. In multicast, you connect to a "group" rather than to a remote address, and other people do the same. Whereas the procedure to open a multicast connection is very similar to that of unicast, there are crucial differences between the two technologies:

- A multicast group address does not correspond to a real machine address. Those are virtual addresses used by the multicast routers to dispatch the messages to the terminal machines that requested a group connection.
- Connecting to a group does not mean that you actually contact somebody. You could be the only member of the group (in which case you are considered as the group creator). Consequently, you never send or receive a message to or from anybody. You just send and receive messages to and from the group itself.

- Unless you explicitly require this information, there is no reason why you should know who are the other group members, or even if there are any of them. Consequently, when you send a message to the group, you don't know a priori who might receive it. Similarly, when you receive a message from the group, you don't know a priori who sent it.

The multicast technology, while recent, is already widely used. The most important field of application is currently audio-video conferencing (see the vic and vat applications), but any kind of groupware application is potentially subject to multicasting.

1.2 Anatomy of a Multicast Group Address

To contact a multicast group, you must know its address. A multicast address is not much different from a standard unicast network address, apart from the presence of an additional field. The standard syntax for a multicast address is "`address/port/ttl`".

- `address` is an IP-like address, ranging from `224.0.0.0` to `239.255.255`. Addresses beginning with 224 and 239 should not be used, as they are reserved for administration purpose. Other addresses are available to the public, which gives about 250 millions possibilities.
- `port` is a usual port number.
- `ttl` stands for "time-to-live". This field defines the scope of the multicast group, or more precisely how far the messages you send will be propagated. This scope can vary from sender-only to world-wide.

Figure 2 presents the possible values and their corresponding scope. As the ttl can vary from sender-only to world-wide, you can potentially have different groups using the same address: indeed, the same address could be used in different countries, with a ttl inferior to 64. In such a case, each group member would only see the other members from the same country, and consequently, the address could be used for totally different things.

1.3 The multicast C API

Here, we would like to draw the outlines of the procedure needed to create a multicast connection, and show how simple it is, notably compared to the standard procedure for creating a point-to-point connection.

In a first step, we have to create the sockets. While in unicast, one socket is enough, you need different sockets for reading and writing on a multicast group, because they will be used slightly differently.

Value	Scope
0	Sender
1	Local Network
16	Site
32	Region
48	Country
64	Continent
128	World

Figure 2: Time-To-Live value/scope correspondence

```
r = socket (..., SOCK_DGRAM, ...);
w = socket (..., SOCK_DGRAM, ...);
```

Next, some setup on the sockets is needed. In unicast, you generally only have to "connect" the socket. In multicast, you must tell the reading socket to receive the messages from the group, and bind it. You must also set the ttl for the writing socket.

```
setsockopt (r, IP_PROTO_IP, IP_ADD_MEMBERSHIP, ...);
bind (r, ...);

setsockopt (w, IP_PROTO_IP, IP_MULTICAST_TTL, ...);
```

Once the connection is established, you will read and write to and from the group. In multicast, which is based on UDP, it is preferable to use the non connected system calls, notably for reading (see the last section).

```
rcvfrom (r, ...);
sendto (w, ...);
```

Finally, when the connection is no longer needed, you close the sockets. In multicast, it is also a good idea to inform the system that the reading socket will leave the multicast group.

```
setsockopt (r, IP_PROTO_IP, IP_DROP_MEMBERSHIP, ...);
close (r);
close (w);
```

As you can see, opening a multicast connection costs only a few more system calls than in unicast. Several options are also available for setting up a multicast connection. Here are the most useful:

- You can allow multiple connections from the same machine by setting the `SOL_SOCKET SO_REUSEADDR` socket option on the reading socket *before* binding it. This actually allow several applications to reuse the same port.
- The `IP_PROTO_IP IP_MULTICAST_LOOP` socket option will allow you to receive your own messages.

2 Multicast Support in XEmacs

XEmacs features a process abstraction that allows you to start external processes and communicate with them. As a special kind of process (which are not actually children of the XEmacs process), TCP stream network connections are already supported, thanks to the `open-network-stream` function. When implementing the multicast support, the idea was obviously to keep as much compatibility as possible with the current process abstraction in order to make multicast support just another particular case of network connection.

2.1 Multicast Group Creation

The work needed to implement the multicast support is actually rather simple. It consists mainly in one internal function plus a Lisp wrapper around it. The internal function is called `open-multicast-group-internal`. This function is written in C in `process.c` and has two purposes: creating the actual network connection (this functionality is split across system-specific files like `process-unix.c`) and associating it with an XEmacs process that will be handled from the Lisp level.

The Lisp wrapper is called `open-multicast-group`, resides in `multicast.el` (which is dumped) and only differs from the internal C function in the required arguments. While the internal function takes the group parameters as separate arguments, the Lisp wrapper takes a single argument describing the connection in the standard multicast form: a Lisp String of the form `"dest/port/ttl"`. Thus, we get a prototype very similar to that of `open-network-stream`:

```
- open-multicast-group (NAME BUFFER ADDRESS)

  Open a multicast connection on the specified address.
  Returns a subprocess-object to represent the connection.
  Input and output work as for subprocesses; 'delete-process'
  closes it. Args are NAME BUFFER ADDRESS.

  NAME is a name for the process. It is modified if necessary
  to make it unique.

  BUFFER is the buffer (or buffer-name) to associate with the
  process. Process output goes at the end of that buffer, unless
  you specify an output stream or filter function to handle the
  output. BUFFER may be also nil, meaning that this process is not
  associated with any buffer.

  ADDRESS specifies a standard multicast address "dest/port/ttl":
  dest is an internet address between 224.0.0.0 and 239.255.255.255
  port is a communication port like in traditional unicast
  ttl is the time-to-live.
```

2.2 Multicast Group Manipulation

After the multicast connection is created, you can manipulate it with just the same functions as for any other XEmacs process. The most important ones are:

- `process-send-string` or `process-send-region`, which will send a string to the group,
- `set-process-filter`, which allows you to manipulate any received message instead of directly inserting it into a buffer,
- for a Mule-ized XEmacs, `set-process-coding-system` along with some input method would allow you to create textual groups in any language,
- and finally, `delete-process` which will close the connection.

3 MChat, the first multicast package for XEmacs

In this section, we would like to present the first (and currently only) XEmacs package using the internal multicast support. MChat stands for "Multicast Chatting". This package allows you exchange textual messages on a multicast group with other participants. Figure 3 shows an XEmacs frame dedicated to an MChat group.

3.1 Description of the package

The purpose of this package is to implement a group conversation facility based on the multicast technology. By calling the `mchat` function, you join a multicast group in which you can send and receive small textual messages (typically less than 500 octets, see the next section). This function sets up an XEmacs frame with 2 MChat buffers dedicated to this group.

The bottom buffer is called the "message buffer". It is in a text edition major mode. This is where you edit your own messages before sending them. An MChat minor mode in this buffer gives you key bindings to send the message, circulate in the old messages cache and perform other operations on the group, like ringing a bell, display the list of known participants, quitting the group...

The top buffer is called the "group buffer". This is where the conversation takes place. You receive messages from the other participants, along with your own messages in this buffer. Each message is prefixed with a "tag" defaulting to the sender's full name (`user-full-name`). You also get "control messages" in this buffer, for instance a line informing you that such person has joined the group.

3.2 The MChat protocol

The protocol used for implementing this functionality is pretty simple, and currently handled entirely in Lisp. This is not really a feature however, but rather a consequence of the current limitations of the process abstraction with respect to multicast connections. Those limitations will be detailed in the next section.

An overview of the MChat communication protocol is given below in pseudo BNF format:

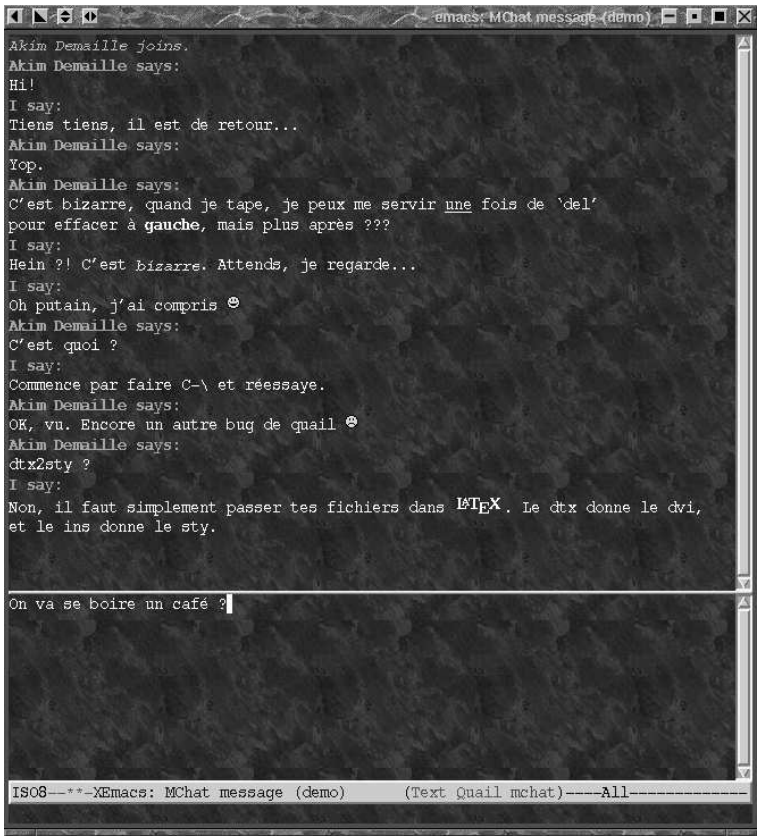

Figure 3: A screenshot of MChat in an XEmacs frame

```
MSG           := PROTO_VERSION USER_ID LENGTH DATA
PROTO_VERSION := MAJOR MINOR
USER_ID       := USER_IP USER_PID
DATA          := ATOM CONTENTS
```

The `USER_ID` part permits to identify uniquely each participant. Please refer to the next section to see why it is currently needed. `LENGTH` gives the length of the following data (a protection against possible messages concatenation), and the data consists of a protocol atom and the atom contents, if any. Figure 4 gives a list of the protocol atoms currently implemented. The `join` atom is used when you arrive on a group. It also provides the other participants with your own tag. The `ring` and `msg` atoms allow you to produce an audio signal or send a message to the group. `whois` might be used to require information about a member for whom you don't have the tag information (which happens if the person was here before you) and this user should then reply with the `iam` atom and provide his tag. A `heart bit` should be sent whenever you have not sent anything else for a certain amount of time. If one participant is quiet for a long enough time, other members will assume that he has died.

Atom	Contents
join	user tag
quit	(none)
whois	user tag
iam	user tag
ring	(none)
msg	text
heart bit	(none)

Figure 4: MChat protocol atoms, version 2.0

Problems and Perspectives

Although already pretty usable, the multicast support is not completely satisfactory. The main problems we are currently facing actually come from the way the original process abstraction was designed. The 2 most important problems are detailed below.

3.3 Identifying the message sender

The multicast technology is designed in a way that a single network connection (namely, a single socket) is used to receive messages from any group members. Although not theoretically required, you probably want to figure out the identity of the sender of each message. Since the XEmacs process abstraction was originally designed to support only TCP streams, the systems calls used to retrieve information from the network work in "connected" mode (`read` or `recv`) and do not allow you to get the identity of the sender. The only way to get an authentic sender identity would be to use `recvfrom` for multicast connections, which is not currently possible without reworking the process abstraction. This is why the MChat protocol currently contains a user identity part, which is not secure since it could be modified at the application level.

3.4 Implementing application protocols

The multicast technology is based on UDP. This means in particular that since this protocol is insecure (you can loose UDP packets), more work needs to be done at the application level. Nowadays, we can see different application level protocols appearing, suited to different purpose. For example, real time applications interested in always having the last information and for which loosing data is not fatal often use RTP and RTCP (Real Time Transport [Control] Protocol). On the other hand, applications like MChat would prefer to ensure that the data is not corrupted, and would probably prefer using RMP (Multicast Reliable Protocol).

In any case, implementing those protocols should definitely not be up to the Lisp layer. Most of the time, multicast messages need to be fragmented in standalone UDP packets, and reconstituted when received. The Lisp layer should only receive full messages that are known *not* to be corrupted, and should not see the details of such or such protocol implementation.

3.5 Reworking the process abstraction

The preceding remarks show that the current process abstraction is too limited to provide a completely satisfactory multicast support, and should be reworked. In the future, we should be able to support not only multicast connection, but any other kind of network connection, and it should not be difficult to add new ones. What we need to do is:

- provide a more flexible way of creating XEmacs processes and associating them with external programs, network connections of different kinds…
- provide different process "backends", that is different methods to send and receive data to and from a network connection. This also includes the ability to manipulate the data before passing it to the Lisp layer.

The recent developments on the ability to integrate dynamically loadable C modules to XEmacs is a good opportunity to start reworking the design of the external processes interface.