Selective Multicast Communication in Distributed Systems

Chen Chen
Department of Computer Science
University of Maryland

October 5, 1992

Abstract

Most current techniques for communications between the software components of a distributed application are limited to one-to-one communication; there is little support for one-to-many or many-to-many communications. We have developed a framework for selective multicast, a mechanism supporting one-to-many and many-to-many communications, where components of an application can communicate with each other. After discussing the overall requirements for a selective multicast environment, we describe our approach to selective multicast. An environment to support selective multicast in distributed system is then described in detail. We demonstrate selective multicast mechanism by providing an application of connecting Unix tools using selective multicast.

1 Overview

Selective multicast, i.e. one-to-many or many-to-many communication, is important in distributed systems. The software components of an application need to communicate with each other. While using remote procedure call for one-to-one request-response communication is widespread, facilities for one-to-many or many-to-many communication are uncommon.

We view a software application as a system of interoperating processes distributed across a heterogeneous network, where each process is implemented by a module, i.e. a collection of individual data and program units. Each module has interfaces bound to one another representing communication channels between the modules.

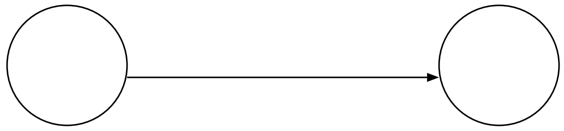

Figure 1: one-to-one communication

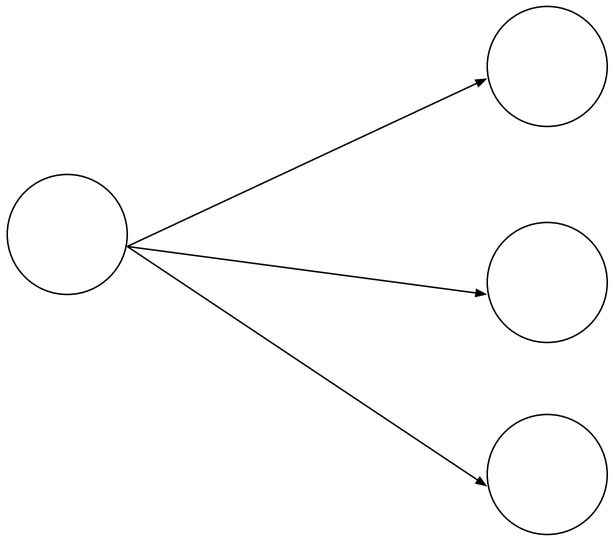

Figure 2: one-to-many communication

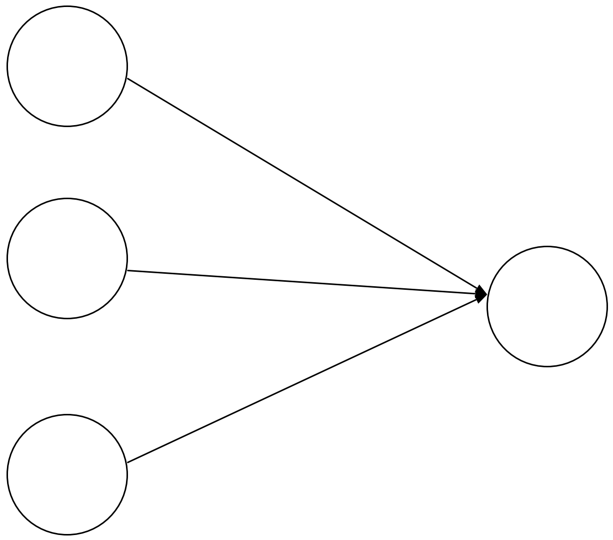

Figure 3: many-to-one communication

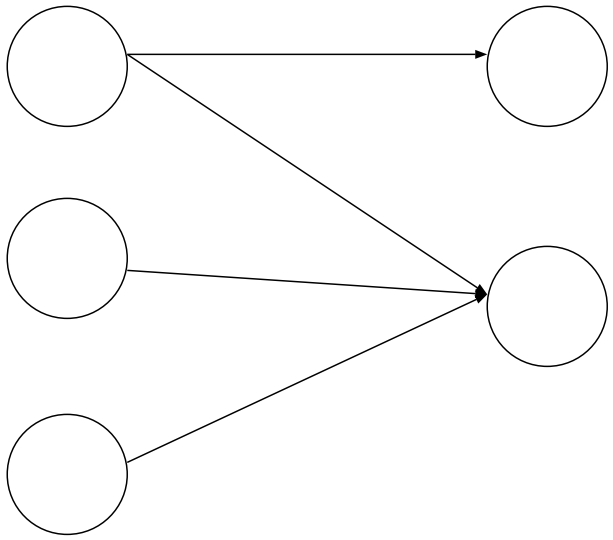

Figure 4: many-to-many communication

Modules need to interact in different ways using interfaces. Figure 1 shows a one-to-one communication structure. We could impletement it by a remote procedure call, or a read-write operation. Figure 2 illustrates a one-to-many communication structure. Here we have one module needs to send messages to multiple modules. A many-to-one communication structure is shown in Figure 3. One module may be interested in messages from more than one module. Figure 4 shows a many-to-many communication structure. One-to-many and many-to-many communications can be supported by selective multicast.

Our research provides a coherent framework for considering all these forms of communication in the presence of system architecture heterogeneity. In Section 2, we motivate a need to build selective multicast facilities. In Section 3, after discussing requirements for a selective multicast environment, we propose a set of primitives. Then in Section 4 we provide an application of connecting Unix tools using selective multicast mechanism.

2 Motivation

This section presents a concrete problem to motivate the selective multicast problem. It will help us to describe the requirements for a selective multicast system.

Figure 5 shows the application structure of an example. In this example, there are four modules **node1, node2, node3** and **node4**, distributed across different host machines.

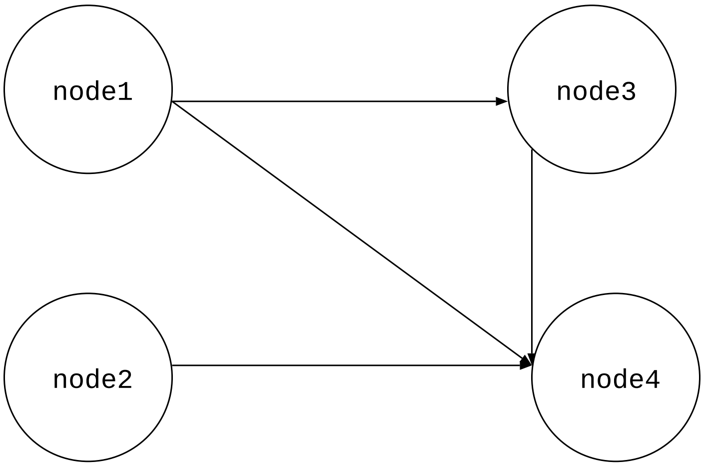

Figure 5: application structure

node3 is interested in messages from **node1**, i.e. **node3** wants to get a certain type of messages sent by **node1** such as *msgtype*13. Similarly **node4** is interested in messages from **node1, node2** and **node3**.

We will illustrate the application shown in Figure 5 in terms of an existing software interconnection system POLYLITH[3]. It represents a software organization where interfacing decisions can be encapsulated separately, using a *software toolbus*. In order to run this example on different hosts in a heterogeneous environment, users need to provide a description of the application's modular structure, in terms of a module interconnection language (MIL). Once this is done, POLYLITH is responsible for invoking processes, and for coercing data representation, synchronization, and marshaling of data during communication.

In the MIL program, we specify information for each module: the location of the executable code, the name of host machine where it should run, and interface information such as either it is a *source* or *sink* and the data type of the messages. In order to put the modules in an application together, we also have to specify bindings between the interfaces to establish communication channels. For example, we use *bind* "*node*1 *send*13" "*node*3 *receive*13" to establish a one way one-to-one communication channel between **node1** and **node3**. When **node1** sends messages using interface *send*13, **node3** can receive these messages using interface *receive*13.

Using asynchronous *read* and *write* one-to-one communication in this example, we find some problems:

```
::::::::::::
example.cl
::::::::::::
service "node1":{
   implementation:{binary: "/users/chenchen/node1.exe"
                   machine: "calloo.cs.umd.edu"}
   source "send13": {string}
   source "send14": {string}
}

service "node2":{
   implementation:{binary: "/users/chenchen/node2.exe"
                   machine: "calvin.cs.umd.edu"}
   source "send34": {string}
}

service "node3":{
   implementation:{binary: "/users/chenchen/node3.exe"
                   machine: "callay.cs.umd.edu"}
   source "send34": {string}
   sink "receive13": {string}
}

service "node4":{
   implementation:{binary: "/users/chenchen/node4.exe"
                   machine: "home.cs.umd.edu"}
   sink "receive14": {string}
   sink "receive24": {string}
   sink "receive34": {string}
}

orchestrate "example":{
   tool "node1"
   tool "node2"
   tool "node3"
   tool "node4"
   bind "node1 send13" "node3 receive13"
   bind "node1 send14" "node4 receive14"
   bind "node2 send24" "node4 receive24"
   bind "node3 send34" "node4 receive34"
}
```

```
:::::::::::
node1.c
:::::::::::
#include <stdio.h>
main (argc, argv)
int argc;
char **argv;
{
   mh_init(&argc, &argv, NULL, NULL);
   mh_write("send13", "S", NULL, NULL, "Hello13");
   mh_write("send23", "S", NULL, NULL, "Hello33");
}

:::::::::::
node2.c
:::::::::::
#include <stdio.h>
main (argc, argv)
int argc;
char **argv;
{
   mh_init(&argc, &argv, NULL, NULL);
   mh_write("send24", "S", NULL, NULL, "Hello24");
}

:::::::::::
node3.c
:::::::::::
#include <stdio.h>
main (argc, argv)
int argc;
char **argv;
{
   char s13[256];

   mh_init(&argc, &argv, NUlL, NULL);
   mh_write("send34", "S", NULL, NULL, "Hello34");
   mh_read("receive13", "S", NULL, NULL, s13);
   printf("s\n", s13);
}

:::::::::::
node4.c
:::::::::::
#include <stdio.h>
main (argc, argv)
int argc;
char **argv;
{
   char s14[256];
   char s24[256];
   char s34[256];

   mh_init(&argc, &argv, NULL, NULL);
   mh_read("receive14", "S", NULL, NULL, s14);
   printf("s\n", s14);
   mh_read("receive24", "S", NULL, NULL, s24);
   printf("s\n", s24);
   mh_read("receive34", "S", NULL, NULL, s34);
   printf("s\n", s34);
   mh_shutdown(0, 42, "");
}
```

Figure 6: MIL program using asynchronous *read* and *write* (left); C source code for each module (right)

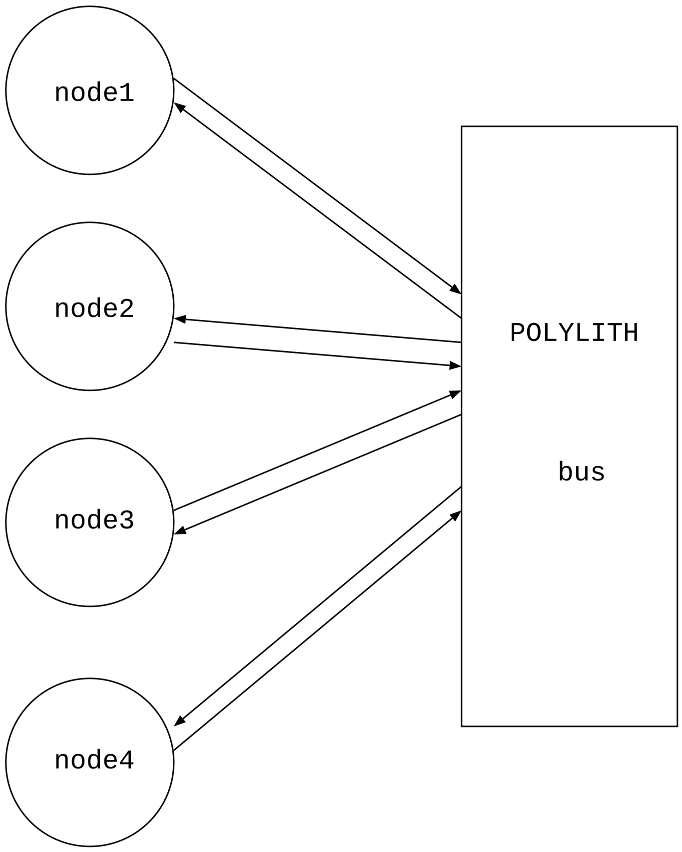

Figure 7: application structure using POLYLITH

- Components must have configuration knowledge. A module must know what modules are interested in messages it sends, i.e. it has to declare explicitly a number of interfaces in its MIL description. Similarly, a module must know what modules will send messages to it, i.e. it has to declare explicitly a number of interfaces in its MIL description. It is hard to reconfigure an application structure. We can not add new modules dynamically.

- In an one-to-many communication, a message will be delivered to the software toolbus many times. This duplicate work increases the cost of the run time performance.

In this application we are assuming that a module can only multicast one type of message. There are cases that a module may need to multicast multiple types of messages. If we allow multiple types of messages to be sent between modules, the situation gets more interesting and complicated. A naive approach to supporting this scenario is to have different *sources* and *sinks* for different type of messages. But it is not reasonable to force modules to know how many types of message there are.

All these problems tell us that using a one-to-one communication mechanism to support selective multicast is not efficient. We need a better facility to support it.

3 Selective Multicast Framework

Our objective is to provide a framework for selective multicast in distributed applications. An environment to support selective multicast must meet the following requirements:

- Users need an easy way to configure and invoke an application.
- The mechanism must not compromise the data type system of the programming language. The parameters must be marshaled correctly. The low level representations of primitive data types on diverse underlying architectures should match.
- A module should be able to declare any type of message.
- A module should be able to register its interest in any type of message.
- A module should be able to multicast any message.
- A module should be able to get a multicast message it is interested in it.
- Selective multicast should be provided at minimum cost to programmers and without loss of run-time performance.

Our approach to meeting the above requirements is to build upon the existing POLYLITH interconnection system mentioned in Section 2. We made this decision because POLYLITH already provides users with an environment that facilitates construction of applications for execution in heterogeneous environments.

POLYLITH satisfies our goal of data type safety. The bus already manages data marshaling. Data is encoded into a stream. When the stream is transmitted to another module, it is decoded into the corresponding data structure. In addition to the data marshaling, the bus coerces the low level representation of primitive data types on different underlying architecture.

The first two requirements for a selective multicast environment can be met by the existing POLYLITH interconnection system. However, our remaining requirements are not answered yet.

Using POLYLITH, the modules interface directly with a software toolbus. This simplifies our implementation of selective multicast. We expand the functionality of the software toolbus to keep track of information about modules interests and communication channels. Modules that want to multicast messages do not have to know which modules are interested in this type of message. Modules that want to get multicast messages do not have to know where the messages come from.

Now we present details concerning the environment we have constructed for experimenting with selective multicast. We have added a set of primitives to POLYLITH to support selective multicast:

- *mh_msgtype*(*data_type*, *msgtype_name*)

  A module uses this service to declare a message type named *msgtype_name* with data type *data_type*. For example, *mh_msgtype*("*S*", "*msgtype*13") declares a selective multicast message type "*msgtype*13" of data type string. This bus service will insert a new message type into an internal table that keeps track of module message type registration.

- *mh_rgsmulticast*(*msgtype_name*)

  A module uses this service to register its interest in message type *msgtype_name*. This bus service will record this information by modifying the corresponding entry in the internal table.

- *mh_multicast*(*msgtype_name*, *data_type*, *message*)

  A module uses this service to multicast a message. This bus service will check the internal table and enqueue a copy of this message on an interface of each module registered in this message type.

- *mh_getmsg*(*data_type*, *msgtype_name*, *message*)

  A module uses this service to receive a message. If the message queue is not empty, this bus service will dequeue a message from the message queue and send it back to the module , otherwise a standard null message will be sent back.

- $mh\_nomsg(message)$

  A module uses this service to declare a standard null message. For example, if the null message is declared as "NO MESSAGE", and a module calls *mh_getmsg* and receives "NO MESSAGE", that means that right now there are no waiting messages. So *mh_getmsg* is nonblocking.

- $mh\_query\_msgtype(msgtype\_buffer)$

  A module uses this service to list all message types in which it has registered an interest. This bus service checks the internal table and sends back the result.

Figure 8 shows what the MIL and C code looks like for the example in Section 2 using our selective multicast primitives. For example, **node3** declares a message type *msgtype*34 causing the service to insert a new message type *msgtype*34 into the internal table. Then it multicasts a string "Hello34" of message type *msgtype*34. Checking the internal table, we find out that **node4** is interested in *msgtype*34 type of message, so a copy of "Hello34" is enqueued on interface of **node4**. **node3** declares a standard null message to be "NO MESSAGE" and then registers its interest in message type *msgtype*13. This bus service marks the entry of row **node3** and column **msgtype13**. When **node3** calls to receive a message, we find out that it is interested in messages of type *msgtype*13 by checking the internal table. If queue is not empty, then a message is dequeued from and sent back to **node3**, otherwise it will get "NO MESSAGE". A snapshot of the internal table for this example is shown in Figure 9.

4 Application: Connecting Unix Tools Using Selective Multicast

In distributed systems in order to allow modules to be run on one machine and display on another, some network protocol must be designed to support communication between modules, one of which is called *server*, and the others are called *clients*[6]. This protocol is also used by the client to send requests to the server for information; it is used by the server to send user input or replies to requests back to the client.

POLYLITH can be viewed as a client-server based software interconnection system. Each module in the application is a client. The software toolbus is the server. A set of

```
::::::::::
example.cl
::::::::::
service "node1":{
   implementation:{binary: "users/chenchen/node1.exe"
                   machine: "calloo.cs.umd.edu"
}

service "node2":{
   implementation:{binary: "users/chenchen/node2.exe"
                   machine: "calvin.cs.umd.edu"
}

service "node3":{
   implementation:{binary: "users/chenchen/node3.exe"
                   machine: "callay.cs.umd.edu"
}

service "node4":{
   implementation:{binary: "users/chenchen/node4.exe"
                   machine: "home.cs.umd.edu"
}

orchestrate "example":{
   tool "node1"
   tool "node2"
   tool "node3"
   tool "node4"
}
```

```
:::::::::::
node1.c
:::::::::::
#include <stdio.h>
main (argc, argv)
int argc;
char **argv;
{
    mh_init(&argc, &argv, NULL, NULL);
    mh_msgtype("S", "msgtype13");
    mh_msgtype("S", "msgtype14");
    mh_multicast("msgtype13", "S", "Hello13");
    mh_multicast("msgtype14", "S", "Hello14");
}
```

```
::::::::::
node2.c
::::::::::
#include <stdio.h>
main (argc, argv)
int argc;
char **argv;
{
    mh_init(&argc, &argv, NULL, NULL);
    mh_msgtye("S", "msgtype24");
    mh_multicast("msgtype24", "S", "Hello24");
}
```

```
:::::::::
node3.c
:::::::::
#include <stdio.h>
main (argc, argv)
int argc;
char **argv;
{
    char msgtypebuf[256];
    char s[256];

    mh_init(&argc, &argv, NULL, NULL);
    mh_msgtype("S", "msgtype34");
    mh_multicast("msgtype34", "S", "Hello34");
    mh_nomsg("NO MESSAGE");
    mh_rgsmulticast("msgtype13");
    mh_getmsg("S", s, msgtypebuf);
    if (strcmp(s, "NO MESSAGE")){
        printf("$s: %s\n", msgtypebuf, s);
    }
    else {
        printf("There is no multicast message. \n");
    }
}
```

```
:::::::::
node4.c
:::::::::
#include <stdio.h>
main (argc, argv)
int argc;
char **argv;
{
    char msgtypebuf[256];
    char s[256];
    int i;

    mh_init(&argc, &argv, NULL, NULL);
    mh_nomsg("NO MESSAGE");
    mh_rgsmulticast("msgtype24");
    mh_rgsmulticast("msgtype34");
    mh_rgsmulticast("msgtype14");
    for (i=1; i<=3; i++)
    {
        mh_getmsg("S", s, msgtypebuf);
        if strcmp(s, "NO MESSAGE") {
            printf(%s: %s\n", msgtypebuf, s);
        }
        else {
            printf("There is no multicast message.\n");
        }
     }
     mh_shutdown(0, 42, "");
}
```

Figure 8: MIL and C code for the example in Section 2 using selective multicast

msg type module	msgtype13	msgtype14	msgtype34	msgtype14	...
node1					
node2					
node3	√				
node4		√	√	√	
...					

Figure 9: The internal table keeps track of module message type registration for the example in Section 2

primitives defines a communication protocol. The server interprets requests from clients. Some requests command the server to manipulate messages, like *mh_multicast*, while others ask the server for information, like *mh_query_msgtype*.

An interesting use of our selective multicasting framework is to connect Unix tools. Suppose we have several windows connected by Unix tools. When an event such as pressing a key on the keyboard is generated in a window, its previously registered *callback proc*, a shell script, is notified. When the *callback proc* is triggered, it takes input from this window, executes the corresponding shell script, and then sends its output to other windows. If the system loads are unbalanced with one host machine overloaded, we can distribute the *callback procs* to run on other hosts.

Figure 10 shows an example of this problem. In this example, we have five circles representing fives windows connected by the Unix tools **awk** and **grep**, denoted by ovals. When an event is generated in window **foo**, its previously registered *callback proc* **unixtool_awk** is triggered, taking input from **foo**, executing a Unix command **awk '{print $2, $1}'**, and then sends the output to **bar1** and **new**. When **bar1** gets an event from **unixtool_awk**, its *callback proc* **unixtool_grep** is also triggered. It takes input from **bar1**, executes a Unix command **grep chen**, and sending the output to **bar**. Similarly, **mygrep** takes input from

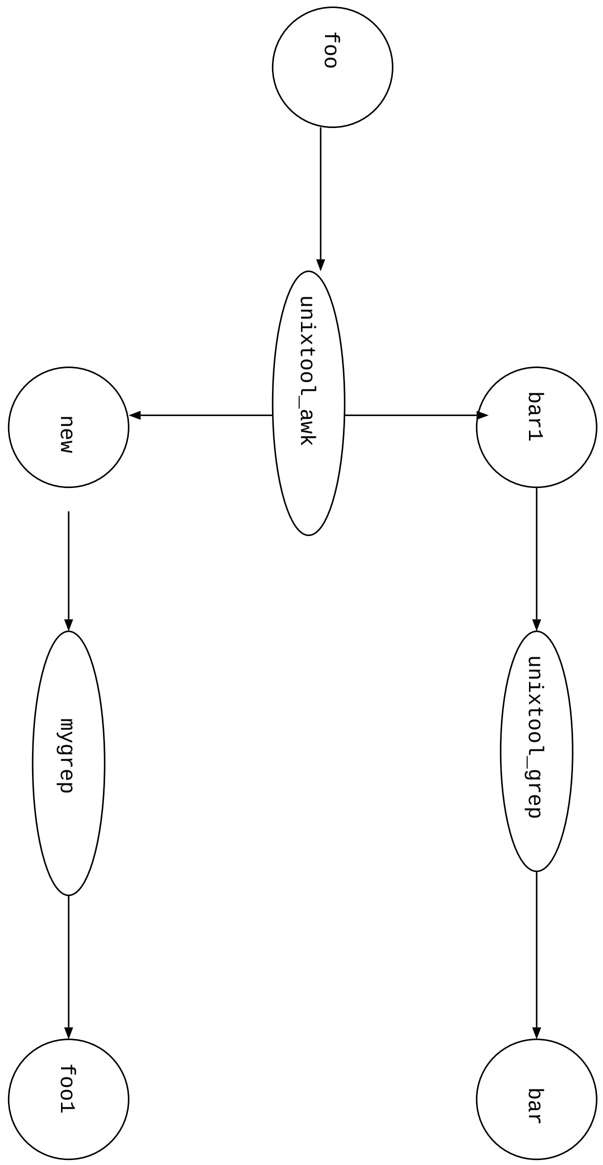

Figure 10: application structure

ident	name	binary	machine	in	command	mode
1	foo1	/users/chenchen/foo1	calvin.cs.umd.edu	mygrep	-	in
2	foo	/flubber/chenchen/foo	flubber.cs.umd.edu	-	-	out
3	bar1	/users/chenchen/bar1	calloo.cs.umd.edu	-	-	inout
4	bar	/users/chenchen/bar	calloo.cs.umd.edu	unixtool_awk	-	in
5	unixtool_awk	./unixtool_awk	callay.cs.umd.edu	foo	awk '{print$2,$1}'	unixtool
6	unixtool_grep	./unixtool_grep	callay.cs.umd.edu	bar1	grep chen	unixtool
7	new	/users/chenchen/new	home.cs.umd.edu	unixtool_awk	-	inout
8	mygrep	./mygrep	home.cs.umd.edu	new	grep jim	unixtool

save quit run

Figure 11: a snapshot of the spreadsheet

new, executes a Unix command **grep jim**, and then sends the output to **foo1**. Multiple modules may be interested in messages from one module. For example, **unixtool_awk** must send output to **bar1** and **new**. We can use our selective multicast mechanism to support this one-to-many communication.

In order to run this application, we need information for modules:

- ident

  identifier of a module

- name

  name of a module

- binary

  location of the binary file of a module

- machine

  host machine name of a module

- in

  names of modules from which it takes input

- command

  if the module is a Unix tool, the shell script will be executed

- mode

  in, out, inout or unixtool

Modes of modules fall into four categories: **in, out, inout or unixtool**. If a module only has input from other modules, for example **foo1** and **bar**, then it is of mode **in**. If a module only has output to other modules(**foo**), then it is of mode **out**. If a module has both input from other modules and output to other modules, it is of mode **inout**(**bar1** and **new**). A module executes a shell script is of mode **unixtool** (**unixtool_awk**, **unixtool_grep** and **mygrep**).

We have implemented this application by building a spreadsheet-like graphical user interface on top of multicasting software toolbus. Figure 11 shows a snapshot of the spreadsheet interface for the example shown in Figure 10. Users do not need to edit C source files

```
::::::::::
in.c
::::::::::
#include <stdio.h>
main (argc, argv)
int argc;
char **argv;
{
    char s[256];
    char msgtypebuf[256]
    char objname_buf[256];
    char pname[256];
    char in[256];
    char *p, *q;

    mh_init(&argc, &argv,  NULL, NULL);
    mh_query_objattr("IN", in, sizeof(in));

    /*decode in into several pnames */
    q = pname;
    for (p = in; p<=(in + sizeof(in)); p++){
        if (*p !=' '){
            while ((*p != ' ') &&
             ((int)p<=(int)(in + sizeof(in))))
                   *q++ = *p++;
            *q = '\0';
            mh_rgsmulticast(pname);
        }
        q = pname;
     }

     /* get message */
     while (1) {
        mh_getmsg("S", s, msgtypebuf);
        while (strcmp(s, "eof\n")){
          if (strcmp(s, "NO MESSAGE"))
            printf("%s: %s\n", msgtypebuf, s);
          mh_getmsg("S", s, msgtypebuf);
        }
     }
}
```

```
::::::::::
out.c
::::::::::
#include <stdio.h>
main (argc, argv)
int argc;
char **argv;
{
    char  objname_buf[256];
    char  buffer[256];

    mh_init(&argc, &argv, NULL, NULL);

    mh_query_objattr("NAME", objname_buf,
                     sizeof(objname_buf));
    mh_msgtype(objname_buf);

    /* read a message from terminal */
    while (1) {
       while(strcmp(gets(buffer), "")){
           strcat(buffer, "\n");
           mh_multicast(objname_buf, "S", buffer);
       }
    }
}
```

Figure 12: C code for module of modes **in**(left) and **out**(right) using selective multicast

```
::::::::::
inout.c
::::::::::
#include <stdio.h>
main (argc, argv)
int  argc;
char **argv;
{
    char s[256];
    char objname_buf[256];
    char pname[256];
    char in[256];
    char *p, *q;

    mh_init(&argc, &argv, NULL, NULL);
    mh_query_objattr("IN", in, sizeof(in));

    /*decode IN into several pnames */
    q = pname;
    for (p=in; p<=(in + sizeof(in));p++){
        if (*p != ' '){
            while ((*p != ' ') &&
             ((int)p<=(int)(in + sizeof(in))))
                   *q++ = *p++;
            *q = '\0';
            mh_rgsmulticast(pname);
        }
        q= pname;
     }

     mh_query_objattr("NAME", objname_buf,
                        sizeof(objname_buf));
     mh_msgtype(objname_buf);

     /* get message */
     while (1) {
        mh_getmsg("S", s, msgtypebuf);
        while (strcmp(s, "eof\n")){
          if (strcmp(s, "NO MESSAGE")){
            mh_multicast(objname_buf, "S", s);
        }
        mh_multicast(objname_buf, "S", "eof\n");
     }
}
```

```
::::::::::
unixtool.c
::::::::::
#include <stdio.h>
{
    char s[256], objname_buf[256], buffer[256],
         command[256], pname[256], name[256],
         in[256], out[256], tempin[256], tempout[256],
          msgtypebuf[256],*p,*q;
    FILE *fpIn, *fpOut;

    mh_init(&argc, &argv, NULL, NULL);
    mh_nomsg("S", "NO MESSAGE");
    mh_query_objattr("COMMAND", command,
                      sizeof(command));
    mh_query_objattr("NAME", name, sizeof(name));
    mh_msgtype(name);
    mh_query_objattr("IN", in, sizeof(in));
    mh_query_objattr("OUT", out, sizeof(out));

    sprintf(tempin, "%stempIn", name);
    sprintf(tempout, "%stempOut", name);

    /* decode IN into several pnames */

    q=pname;
    for (p=in;p<=(in+sizeof(in));p++){
        if (*p!=' '){
           while ((*p!=' ') &&
             ((int)p<=(int)(in + sizeof(int)))){
                 *q++ = *p++;
           *q='\0';
           mh_rgsmulticast(pname);
        }
        q=pname;
    }

    while (1) {
       fpIn =fopen(tempin, "w");
       mh_getmsg("S", s, msgtypebuf);
       while (strcmp(s, "eof\n"){
         if (strcmp(s, "NO MESSAGE"))
             fputs(s, fpIn);
         }
         mh_getmsg("S",s, msgtypebuf);
       }

       fclose(fpIn);
       strcat(command,tempin);
       strcat(command, " > ");
       strcat(command, tempout);
       system(command);

       fpOut = fopen(tempout, "r");
       while ((fgets(buffer, sizeof(buffer),fpOut)!=NULL)
          mh_multicast(name, "S", buffer);
       mh_multicast("S", "eof\n");
    }
}
```

Figure 13: C code for module of modes **inout**(left) and **unixtool**(right) using selective multicast

and Makefile. All they need to do is to provide attributes associated with each module. The source code for a module of each mode shown in Figure 12 and Figure 13 is provided by the tool. For example, in the first row of the spreadsheet, we have a module named **foo1** and the executable of this module is */users/chenchen/foo1* on machine *calvin.cs.umd.edu*. Since **foo1** takes input from module **mygrep** but does not send output to other modules, its mode is **in**. Given the spreadsheet defined by users, information about modules is stored in a file *application_file* by pressing button **save**. When button **run** is pressed, a copy of the executable generated by compiling *in.c* is sent to machine *calvin.cs.umd.edu* with full path name of */users/chenchen/foo1*. At the same time, actions are taken similarly to other modules given in the spreadsheet. After installation, the software toolbus is started up with input file *application_file*.

The source code for modules of each mode use the primitives described in Section 3. For example, in the fifth row of the spreadsheet, we have a module named **unixtool_awk** of mode **unixtool**, so the source code of **unixtool_awk** is *unixtool.c*. In *unixtool.c* (see Figure 13), a call to *mh_rgsmulticast*() registers its interest in all modules given by attribute **in** in the spreadsheet(**foo**). Calls to *mh_getmsg*() are used to receive messages from those modules. When **unixtool_awk** receives a special end of file message, it stores these messages in a *tempIn* file, executes a shell script given by attribute **command**(**awk '{print $2, $1}'**) in the spreadsheet on file *tempIn*, redirects the output to file *tempOut* and then calls *mh_multicast*() to send out the output to **bar1** and **new**. Similar actions are taken to modules of mode **in**, **out** and **inout**. Notice that there might be multiple modules interested in messages from this module, so we have to use selective multicast to support this one-to-many communication.

A feature of this tool is that it is very easy to reconfigure the application structure. Users only need to change the attributes on the spreadsheet. For example, if a user wants the application structure shown in Figure 14, they can achieve this goal by simply changing the spreadsheet to a new one shown in Figure 15.

5 Performance

Our experience to date are that use of the POLYLITH bus organization does not necessarily result in performance loss.

Using the selective multicast primitives built on POLYLITH, the cost of registering

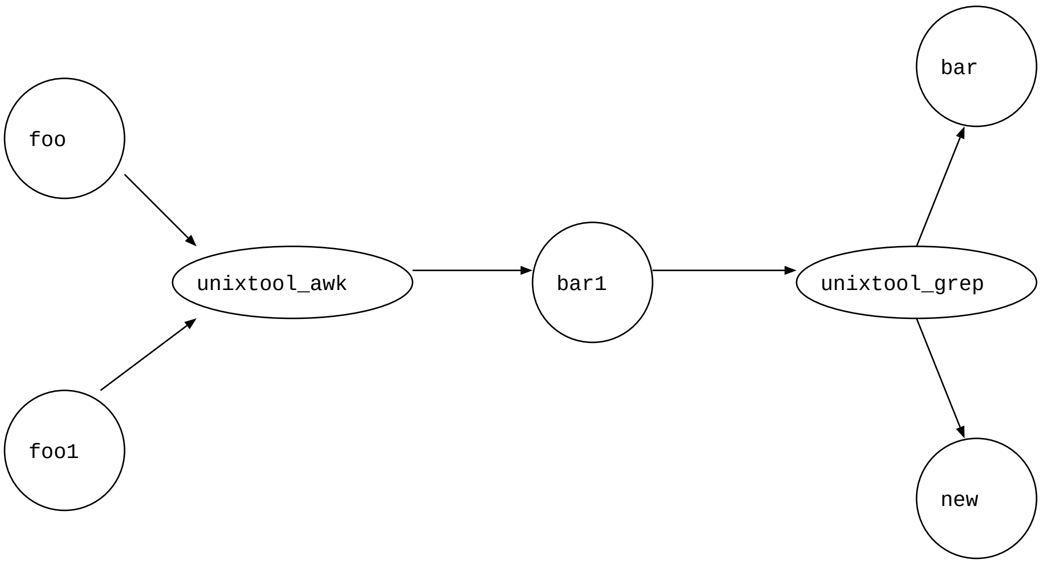

Figure 14: a new application structure

messages and receiving messages is the same as the cost of reading messages in the one-to-one communication. When multicasting a message, in addition to the cost of writing a message, there is a cost in searching the internal table and enqueuing a copy of the message for each module interested in it, which is dependent on the number of modules interested in it.

6 Related Work

Our approach is based upon the software bus abstraction as currently implemented in the POLYLITH interconnection system[3]. We benefited from Cooper's research[1] on programming language support for multicast communication, which discusses essential and desirable properties of a language construct for multicast communication. Reiss's Field environment [5] connects tools with selective broadcasting, allowing the Unix philosophy of letting independent tools cooperate through simple conventions. His work motivates the need for selective multicast and inspired our work on connecting Unix tools using selective multicast.

ident	name	binary	machine	in	command	mode
1	foo1	/users/chenchen/foo1	calvin.cs.umd.edu	-	-	out
2	foo	/flubber/chenchen/foo	flubber.cs.umd.edu	-	-	out
3	bar1	/users/chenchen/bar1	calloo.cs.umd.edu	-	-	inout
4	bar	/users/chenchen/bar	calloo.cs.umd.edu	unixtool_awk	-	in
5	unixtool_awk	./unixtool_awk	callay.cs.umd.edu	foo foo1	awk '{print$2,$1}'	unixtool
6	unixtool_grep	./unixtool_grep	callay.cs.umd.edu	bar1	grep chen	unixtool
7	new	/users/chenchen/new	home.cs.umd.edu	unixtool_grep	-	in

save quit run

Figure 15: a snapshot of the spreadsheet for a new application

7 Conclusion

We have described a broad framework that supports selective multicast, specifically within a distributed programming environment. In order to run experiments within this framework, we have constructed selective multicast facilities upon existing software interconnection system POLYLITH. We also provided an applicaiton of connecting Unix tools using our selective multicast mechanism. This paper exposes our overall approach to selective multicast. We plan to provide more realistic applications built within our environment.

References

[1] Eric C. Cooper. Programming Language Support for Multicast Communication in Distributed System, *IEEE Transactions on Computers*, July 1990.

[2] James M. Purtilo and Christine R. Hofmeister. Dynamic Reconfigurations of Distributed Programs, *The 11th International Conference on Distributed Computing Systems*, May 1991.

[3] James M Purtilo. The Polylith Software Bus.

[4] James M. Purtilo, Christine R. Hofmeister and Joanne Atlee. Writing Distributed Programs in Polylith, *Dept of Computer Science, University of Maryland, CS-TR-2575*, December 1990.

[5] Steven P. Reiss. Connecting Tools Using Message Passing in the Field Environment, *IEEE Transaction on Computers*, July 1990.

[6] Dan Heller. XView Programming Manual, March 1992.