Group Communication Service for DECK

Ricardo Cassali* Marcos Barreto† Rafael Ávila‡ Philippe Navaux§

Institute of Informatics
Federal University of Rio Grande do Sul - UFRGS
Av. Bento Gonçalves, 9500 — Bloco IV
PO Box 15064 — 90501-910 Porto Alegre — Brazil
E-mail: {cassali,barreto,avila,navaux}@inf.ufrgs.br

Abstract

Handling groups of processes is a desirable characteristic in many parallel and distributed applications, being used for providing performance and availability. In this paper we present the group communication service developed for the DECK programming environment, showing how it was implemented and analyzing its performance.

Keywords: group communication, distributed environments, cluster computing.

1 Introduction

For a great number of parallel and distributed applications, groups of processes are frequently needed to provide collective communication facilities in order to allow a set of distributed tasks to exchange data. This kind of facility is supported by communication group mechanisms that offer resources for an application to control a set of processes that are concurrently running within a group context.

DECK (*Distributed Execution and Communication Kernel*) [BAR98, BAR00] is a library for parallel programming on clusters of PCs that foresees the availability of a set of specialized services, among them group communication, to be used as demanded by the applications.

The major contribution of the work presented in this paper is the design and implementation of a group communication service for the DECK library. The goal of this service is to serve as a basic set of functions to allow the user to create its own protocol for collective communication. In this way, this basic service provides primitives for group management and communication among the group members.

This paper is organized as follows: section 2 presents some related works in terms of programming libraries that provide group communication; section 3 presents the DECK environment, showing its internal organization and provided resources; section 4 presents and explains the group communication

*Undergraduate student at UFRGS (CNPq fellow)
†PhD. student, M.Sc. in Computer Science (PPGC/UFRGS, 2000)
‡PhD. student, M.Sc. in Computer Science (PPGC/UFRGS, 1999)
§PhD. (Grenoble, France), Professor at PPGC/UFRGS

service; in section 5 it is shown the performance results obtained using the proposed service, and finally section 6 presents our conclusions and current directions.

2 Related work

Although libraries for parallel and distributed programming are common nowadays, the furnishment of group communication are not deeply explored by these libraries as expected. Due to this reason, a small number of communication libraries provide this facility, such as MPI and PVM.

There are several implementations of the MPI (*Message Passing Interface*) standard [MPI94], such as MPICH [WOR99] and LAM [OHI96], which are currently used to the development of parallel and distributed applications.

The MPI standard defines a set of collective communication primitives that are used for message exchange within a group of MPI processes, where all processes belong to this group. These primitives use the concept of *communicators*, which defines the set of processes involved in a collective communication. Besides, each node is identified by a *rank* and the user can specify a node to send a message providing its rank. Figure 1 shows the functions used for group communication in MPI.

```
int MPI_Bcast(*buf, count, datatype, root, comm);
int MPI_Scatter(*sbuf, scount, stype, *rbuf, rcount, rtype, root, comm);
int MPI_Gather(*sbuf, scount, stype, *rbuf, rcount, rtype, root, comm);
int MPI_Allgather(*sbuf, scount, stype, *rbuf, rcount, rtype, comm);
int MPI_Barrier(comm);
```

Figure 1: MPI collective communication functions.

These functions are executed within the context of the communicator passed as a parameter (*comm* in the examples). All functions have buffers for sending/receiving data; however, in the case of the broadcast function, there is only one buffer, as the process will only send or receive data. The identification of which process will send data in the broadcast and scatter functions is the *root* parameter; it also indicates the process that will receive data on a gather function. There are also parameters for specifing the data type and how many items of this data type will be transmitted.

The `MPI_Bcast` function sends data to all members of a specific group. `MPI_Scatter` splits a message and sends its parts to the group members; whereas `MPI_Gather` does the opposite: it collects parts of a message from the group members and pass them to the root process. `MPI_Allgather` is similar to `MPI_Gather` , however all processes receive the whole message. The `MPI_Barrier` function blocks the caller until all group members have called it; then the call returns at any process only after all group members have entered the call.

PVM (*Parallel Virtual Machine*) [GEI94] also provides several functions for group communication, as shown in Figure 2.

```
int pvm_bcast(char *group, int msgtag);
int pvm_mcast(int *tids, int ntask, int msgtag);
int pvm_gather(int count, int datatype, int msgtag, char *group,
            int root);
int pvm_scatter(void *result, void *data, int count, int dtype,
            int msgtag, char *group, int root);
int pvm_reduce(void *func, void *data, int count, int dtype,
            int msgtag, char *group, int root);
int pvm_barrier(char *group, int count);
```

Figure 2: PVM group communication primitives.

The function `pvm_bcast` asynchronously broadcasts a message stored in the active send buffer to all members of the group. The content of the message can be distinguished by the parameter *msgtag*. In PVM 3.2 and later, the broadcast message is not sent back to the sender.

`pvm_mcast` multicasts a message stored in the active send buffer to *ntask* tasks specified in the *tids* array. The message is not sent to the caller even if its tid is in *tids*. As in the broadcast, the content of the message can be distinguished by the parameter *msgtag*.

In the function `pvm_gather` , a specific member of the group receives messages from each member of the group and gathers these messages into a single array. The function `pvm_scatter` is used to send to each member of a group a section of an array from a specific member of the group. These functions are frequently used to distribute work to and collect results from a group of tasks within a PVM application.

The function `pvm_reduce` performs global operations, such as `min, max` and `sum` over all tasks in a group. All group members call this primitive with their local data and the result of the reduction operation appears on the root task *root*, identified by its tid within the group.

The function `pvm_barrier` blocks the calling process until *count* members of the group have called it. The *count* argument is required because, as PVM supports dynamically group management, it is possible the non-participation of a member task in a `pvm_barrier()` call, since the number of tasks to synchronize is passed as a parameter.

An important characteristic of PVM is that is possible for a task not belonging to a group to use collective communication functions for send and receive data. For example, a PVM task can broadcast a message to all members of a group, even though it may not belong to that group.

Besides these functions, both PVM and MPI provide functions for group management, allowing the user to specify the creation and destruction of a group, as well as include or remove members to/from a group.

3 The DECK environment

DECK (*Distributed Execution and Communication Kernel*) is a library for parallel and distributed programming over cluster-based architectures. It is composed of a runtime system and an user API which provides a set of abstractions and more specialized services used as demanded by the applications. A DECK application runs in a SPMD style, in which there is a copy of the application process for each node in the cluster.

Internally, DECK is divided in two layers, one called μDECK, which directly interacts with the underlying hardware and operating system, and a service layer, where more elaborate facilities (including group communication) are made available. Figure 3 shows the layered structure of DECK.

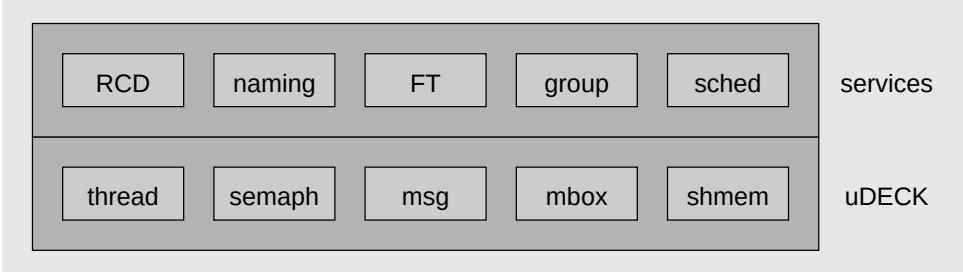

Figure 3: Internal structure of DECK.

μDECK is the platform-dependent part of DECK. This layer implements the basic abstractions provided within the environment: *threads*, *semaphores*, *messages*, *mailboxes* and *shared segments*. Each of these abstractions is treated by the application as an object and has associated primitives for proper manipulation.

Messages can be posted to or retrieved from mailboxes. Only the creator of a mailbox is allowed to retrieve messages from it, but any other thread knowing the mailbox can post to it. To use a mailbox, the creator must register it in a naming server. There are two ways to obtain a mailbox address: fetching it in the name server or receiving it in a message.

Regarding the naming service of the DECK´s upper layer, there is a name server running as a dedicated thread on the first node within the cluster, which is responsible for registering mailboxes. The name server is automatically executed when the application starts and has a well-known mailbox to allow other threads to communicate with it.

The upper layer is responsible to provide a set of services which are platform-independent and implemented on top of the basic abstractions provided in the μDECK layer. This layer foresees the implementation of services such as fault tolerance, load balancing, inter-cluster communication (RCD) and group communication. In this scope of this work, only the latter will be discussed.

The main goal in the design of DECK was use it as the communication library for DPC++ [SIL00], which is a distributed object-oriented language that will use the group communication service from

DECK to provide load balancing and visualization facilities. With the constant use of clusters as platforms for parallel and distributed programming, we have changed the design of DECK in order to additionally provide resources for general use.

The current implementation of DECK runs on Unix, using Pthreads [IEE95] and a socket-based mechanism for communication. Future implementations will run over different network devices, such as Myrinet [BOD95] and SCI [IEE92] and will use communication protocols, such as BIP [PRY98] and SISCI [GIA98], specific for these networks. The shared memory abstraction is not currently provided by DECK, since it depends on the implementation on top of SCI.

4 The group communication service

The group communication service has a Client/Server semantic. There is a server for each group who keeps data related to the group context and executes operations requested by the members (clients) of the group. In this way, a member of the group does not send or receive data directly to another member; all communication in the group context is done by the group server.

Figure 4 shows the group communication functions for DECK, which are explained in the next sections.

```
int deck_group_create (deck_group_t *g, char *g_name, int min);
int deck_group_destroy (deck_group_t *g);
int deck_group_join (deck_group_t *g, deck_mbox_t *mb, char *name);
int deck_group_leave (deck_group_t *g);
int deck_group_bcast (deck_group_t *g, deck_msg_t *msg);
int deck_group_sync (deck_group_t *g);
```

Figure 4: DECK functions for group communication.

4.1 Group creation and destruction

When a group is created, a local thread is launched on the machine from where `deck_group_create()` function was called. This thread is the group server and it holds a list of mailboxes from processes or threads that belong to this group. There is no message passing in the group creation process, as it involves just the creation of a new thread. The group server creates a mailbox and binds it with the name of the group to be able to receive service requests. The parameters are the group object (that will be returned to the new member), the name of the group (to allow others processes/threads to join the group) and the minimum number of members the group must have to start. This last parameter is need to avoid broadcasts before all members have entered the group.

When the `deck_group_destroy()` function is called, a message requesting the end of that group is sent to the group server. The group server waits all broadcasts to be done and exits, returning a message

to the one who requested the group destruction to indicate the success of the operation. It works like join and leave functions, requiring only two messages. The only parameter this function has is the group object that represents the group to be destroyed.

4.2 Join and leave operations

Using the group name, the `deck_group_join()` function fetches the group server mailbox and posts a join request to it. The mailbox passed as parameter will be added in the group server list of mailboxes. The server returns an identification number for that member through the group object. The join function involves passing two messages, as shown in figure 5. The mailbox passed as a parameter is the one to where broadcasts will be sent. In this way, broadcasted messages can be retrieved normally from that mailbox, using the basic DECK function `deck_mbox_retrv()`.

The `deck_group_leave()` function sends a leave request to the group server, with the identification number from the group object passed as parameter. The group server removes the mailbox associated with that *id* number and returns a code for the success of this operation. This is similar to the `deck_group_join()` function as it also involves just two messages (figure 5).

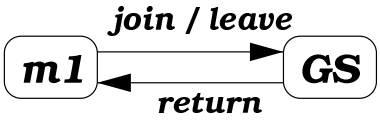

Figure 5: Schematic of join and leave functions.

4.3 Broadcast

The join/leave functions only add or remove mailboxes from the list mantained on the group server. When a broadcast is requested, another thread on the same node where the group server is running is created to handle it. This is done to avoid locking the group server during the broadcast. To execute this function, it will be passed `1 + number of members` messages, as shown in figure 6. This function has as parameters the group object to which the message will be sent and the message itself.

4.4 Synchronization

During a `deck_group_sync()` function, the calling thread communicates with the group server. The group server marks the corresponding mailbox as waiting for synchronization and checks the other ones. When all the mailboxes on the list were marked, it releases all group members by returning a message to each of them. The number of messages passed on synchronization is `2 x number of members`, as shown in figure 7.

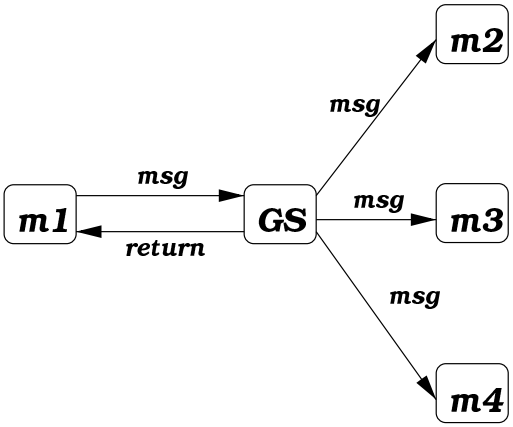

Figure 6: Message exchange for broascast.

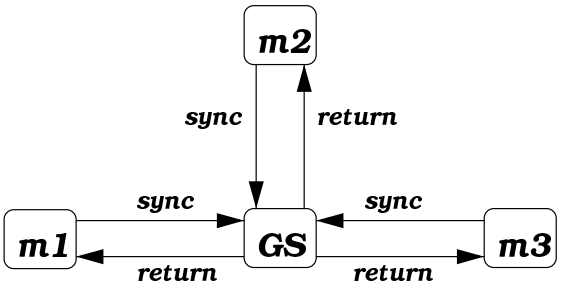

Figure 7: Message passing during synchronization.

5 Performance evaluation

This section shows a preliminary performance evaluation of the proposed group communication service.

The analysis was made in a 4-node PC based cluster connected by the Fast Ethernet communication network. Each node is a Dual Pentium Pro at 200 MHz with 64 MB of main memory. The operating system is Linux, kernel 2.2.10 with `gcc` version 2.91.60 (egcs-1.1.1).

One test performed with this service was the evaluation of the broadcast function. It was made to illustrate the impact caused by the number of members in a group on the latency of the broadcast function. The measured results are presented in figure 8. As comparison, the same test was performed for the MPICH, version 1.1.2.

To generate the presented results, the program calls the function `deck_group_bcast()` followed by the function `deck_group_sync()` 500 times, and then calculates the balanced mean. Time has

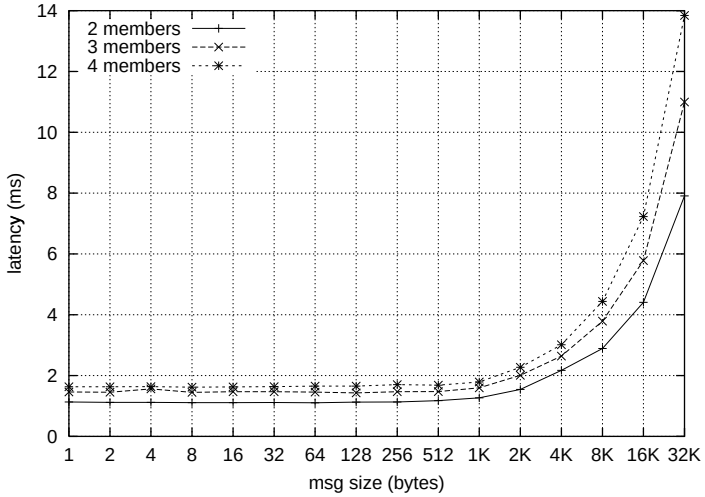

Figure 8: Latency for `deck_group_bcast()`.

been measured with `gettimeofday()` function. It was needed to synchronize the group to marking the time as DECK uses threads to realize broadcasts. The program was run with messages varying from 1 to 32768 bytes. For the MPICH program, the equivalent functions `MPI_Bcast()` and `MPI_Barrier()` were used.

As shown in figure 8, the performance of `deck_group_bcast()` in conjunction with the function `deck_group_sync()` is pratically linear for messages with more than 1 KB, depending on the number of members in the group and on message size.

We can see in figure 9 that DECK has a reasonable performance when compared with MPICH — around of 0.5 ms slower for messages until 2 KB and around of 1 ms for larger messages. This is due two reasons: i) DECK centralizes all broadcast messages in a server who keeps the group status and ii) each time a broadcast is required, a new thread is created and it will not run until getting the processor on the group server.

6 Conclusion

This work has presented the development of a group communication service that was planned to the DECK library, in order to extends its suitability to support different kinds of applications, specially the ones involving data parallelism.

Regarding the DPC++ language, the group communication service is currently used as a basic protocol to the development of load balancing mechanisms, needed each time a distributed object must be allocated within the cluster; and to support debug and visualization facilities. Also, DECK will use this

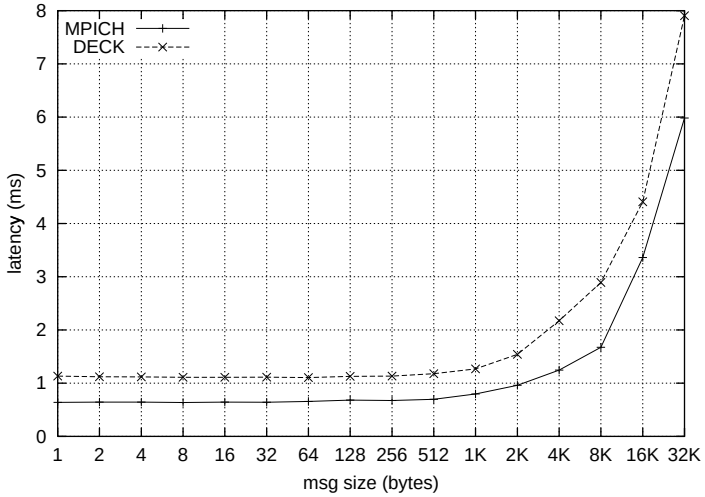

Figure 9: DECK and MPICH latencies for a 2 member group.

service to develop a cluster management service, allowing the user to have a centralized view of the cluster's resources.

The achievable performance of the service was quite satisfactory when compared to MPI, even regarding its multithreaded behaviour, which tends to increase the overhead during message broadcasts since new threads must be created. Also, MPI has presented some problems with a 4-member group that deny more accurate tests.

The current work involves a "fine-tuning" in the group communication protocol, in order to increase its performance, and the addition of more elaborated collective communication functions, such as `scatter` and `gather` . Regarding DPC++, the load balancing mechanism is in development using this basic group communication protocol to allow the distributed objects to exchange data about their workloads during the object creation process.

References

[BAR00] BARRETO, Marcos; ÁVILA, Rafael; NAVAUX, Philippe. The MultiCluster model to the integrated use of multiple workstation clusters. In: WORKSHOP ON PERSONAL COMPUTER BASED NETWORKS OF WORKSTATIONS, 3., 2000, Cancun. **Proceedings...** Berlin: Springer, 2000. p.71–80. (Lecture Notes in Computer Science, v.1800).

[BAR98] BARRETO, Marcos E.; NAVAUX, Philippe O. A.; RIVIÈRE, Michel P. DECK: a new model for a distributed executive kernel integrating communication and multithreading for support

of distributed object oriented application with fault tolerance support. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN, 4., 1998, Neuquén, AR. **Anales...** Neuquén: Universidad Nacional de Comahue, Facultad de Economia y Administración, Departamento de Informática y Estadística, 1998. v.2, p.623–637.

[BOD95] BODEN, N. et al. Myrinet: a gigabit-per-second local-area network. **IEEE Micro**, Los Alamitos, v.15, n.1, p.29–36, Feb. 1995.

[GEI94] GEIST, Al et al. **PVM**: parallel virtual machine. Cambridge, MA: MIT Press, 1994.

[GIA98] GIACOMINI, F. et al. **Low-level SCI software requirements, analysis and predesign**. [S.l.]: ESPRIT Project 23174 — Software Infrastructure for SCI (SISCI), 1998.

[IEE92] INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. **IEEE standard for scalable coherent interface (SCI)**. IEEE 1596-1992, 1992.

[IEE95] INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. **Information technology—portable operating system interface (POSIX), threads extension [C language]**. IEEE 1003.1c-1995, 1995.

[MPI94] MPI FORUM. **The MPI message passing interface standard**. Knoxville: University of Tennessee, 1994.

[OHI96] OHIO SUPERCOMPUTER CENTER. **MPI primer/developing with LAM**. [S.l.]: Ohio State University, 1996.

[PRY98] PRYLLI, Loïc; TOURANCHEAU, Bernard. BIP: a new protocol designed for high performance networking on Myrinet. In: IPPS/SPDP'98 WORKSHOPS, 10., 1998. **Proceedings...** Springer, 1998. p.472–485. (Lecture Notes in Computer Science, v.1388).

[SIL00] SILVEIRA, André et al. DPC++: object-oriented programming applied to cluster computing. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, 2000, Las Vegas. **Proceedings...** Las Vegas: CSREA Press, 2000. p.2515–2521.

[WOR99] WORRINGEN, Joachim; BEMMERL, Thomas. MPICH for SCI-connected clusters. In: SCI-EUROPE, 1999, Toulouse, France. **Proceedings...** 1999. p.3–11.