ASSIGNMENT NO. 4

AIM: Design a distributed application using Message Passing Interface (MPI) for remote computation where client submits a string to the server and server returns the reverse of it to the client.

PRE REQUSITE: Concept of Client Server Communication & Working of Conventional Procedures.

THEORY:

MPI Introduction:

Message Passing Interface (**MPI**) is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in different computer programming languages such as Fortran, C, C++ and Java. There are several well-tested and efficient implementations of MPI, including some that are free or in the public domain. These fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications.

- A message-passing library specifications:
    - Extended message-passing model
    - Not a language or compiler specification
    - Not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks.
- Communication modes: *standard*, *synchronous*, *buffered,* and *ready*.
- Designed to permit the development of parallel software libraries.
- Designed to provide access to advanced parallel hardware for
    - End users
    - Library writers
    - Tool developers

MPI Architecture:

MPI has three main abstraction layers,

- *Open, Portable Access Layer (OPAL)*: OPAL is the bottom layer of Open MPI's abstractions. Its abstractions are focused on individual processes (versus parallel jobs). It provides utility and glue code such as generic linked lists, string manipulation, debugging controls, and other mundane—yet necessary—functionality.
  OPAL also provides Open MPI's core portability between different operating systems, such as discovering IP interfaces, sharing memory between processes on the same server, processor and memory affinity, high-precision timers, etc.
- *Open MPI Run-Time Environment (ORTE)* (pronounced "or-tay"): An MPI implementation must provide not only the required message passing API, but also an accompanying run-time system to launch, monitor, and kill parallel jobs. In Open MPI's case, a parallel job is comprised of one or more processes that may span multiple operating system instances, and are bound together to act as a single, cohesive unit.
  In simple environments with little or no distributed computational support, ORTE uses rsh or ssh to launch the individual processes in parallel jobs. More advanced, HPC-dedicated environments

typically have schedulers and resource managers for fairly sharing computational resources between many users. Such environments usually provide specialized APIs to launch and regulate processes on compute servers. ORTE supports a wide variety of such managed environments, such as (but not limited to): Torque/PBS Pro, SLURM, Oracle Grid Engine, and LSF.

- *Open MPI (OMPI)*: The MPI layer is the highest abstraction layer, and is the only one exposed to applications. The MPI API is implemented in this layer, as are all the message passing semantics defined by the MPI standard.
  Since portability is a primary requirement, the MPI layer supports a wide variety of network types and underlying protocols. Some networks are similar in their underlying characteristics and abstractions; some are not.

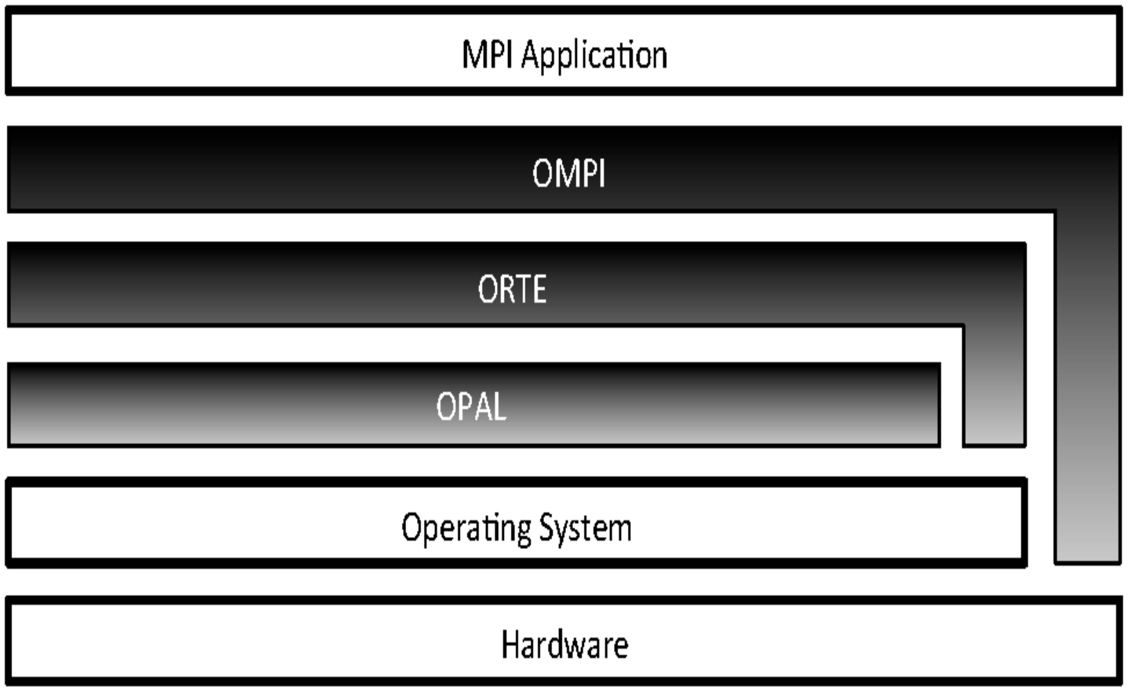


Figure 1: layer architectural view of MPI showing its three main layers: OPAL, ORTE, and OMPI

Although each abstraction is layered on top of the one below it, for performance reasons the ORTE and OMPI layers can bypass the underlying abstraction layers and interact directly with the operating system and/or hardware when needed. For example, the OMPI layer uses OS-bypass methods to communicate with certain types of NIC hardware to obtain maximum networking performance.

Each layer is built into a standalone library. The ORTE library depends on the OPAL library; the OMPI library depends on the ORTE library. Separating the layers into their own libraries has acted as a wonderful tool for preventing abstraction violations. Specifically, applications will fail to link if one layer incorrectly attempts to use a symbol in a higher layer. Over the years, this abstraction enforcement mechanism has saved many developers from inadvertently blurring the lines between the three layers.

MPI Features:

- General
  -- Communicators combine context and group for message security
  -- Thread safety
- Point-to-point communication
  -- Structured buffers and derived datatypes, heterogeneity
  -- Modes: normal (blocking and non-blocking), synchronous, ready (to allow access to fast protocols), buffered
- Collective
  -- Both built-in and user-defined collective operations
  -- Large number of data movement routines
  -- Subgroups defined directly or by topology

MPI Library:

The MapReduce-MPI (MR-MPI) library, which is an open-source implementation of MapReduce written for distributed-memory parallel machines on top of standard MPI message passing.
MapReduce is the programming paradigm, popularized by Google, which is widely used for processing large data sets in parallel. Its salient feature is that if a task can be formulated as a MapReduce, the user can perform it in parallel without writing any parallel code. Instead the user writes serial functions (maps and reduces) which operate on portions of the data set independently. The data-movement and other necessary parallel operations can be performed in an application-independent fashion, in this case by the MR-MPI library.
The MR-MPI library was developed at Sandia National Laboratories, a US Department of Energy facility, for use on informatics problems. It includes C++ and C interfaces callable from most hi-level languages, and also a Python wrapper and our own OINK scripting wrapper, which can be used to develop and chain MapReduce operations together. MR-MPI and OINK are open-source codes, distributed freely under the terms of the modified Berkeley Software Distribution (BSD) License. The authors of the library are Steve Plimpton and Karen Devine.

MPI Sample Code:

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
        const int tag = 42;     /* Message tag */
        int id, ntasks, source_id, dest_id, err, i;
        MPI_Status status;
        int msg[2]; /* Message array */
        err = MPI_Init(&argc, &argv); /* Initialize MPI */
        if (err != MPI_SUCCESS)
        {
                printf("MPI initialization failed!\n");
                exit(1);
        }
        err = MPI_Comm_size(MPI_COMM_WORLD, &ntasks); /* Get nr of tasks */
        err = MPI_Comm_rank(MPI_COMM_WORLD, &id);  /* Get id of this process */
        if (ntasks < 2)
        {
                printf("You have to use at least 2 processors to run this program\n");
                MPI_Finalize();   /* Quit if there is only one processor */
                exit(0);
        }
        if (id == 0)
        {  /* Process 0 (the receiver) does this */
                for (i=1; i<ntasks; i++)
                {
                        err = MPI_Recv(msg, 2, MPI_INT, MPI_ANY_SOURCE, tag,
                                MPI_COMM_WORLD, &status);          /* Receive a message */
                        source_id = status.MPI_SOURCE; /* Get id of sender */
                        printf("Received message %d %d from process %d\n", msg[0], msg[1],
                        source_id);
                }
        }
        else
        {    /* Processes 1 to N-1 (the senders) do this */
                msg[0] = id; /* Put own identifier in the message */
                msg[1] = ntasks;        /* and total number of processes */
                 dest_id = 0; /* Destination address */
                err = MPI_Send(msg, 2, MPI_INT, dest_id, tag, MPI_COMM_WORLD);
        }
        err = MPI_Finalize();         /* Terminate MPI */
        if (id==0)
                printf("Ready\n");
        exit(0);
        return 0;
}
```

How MPI works ?

MPI stands for Message Passing Interface. It is a message-passing specification, a standard, for the vendors to implement. In practice, MPI is a set of functions (C) and subroutines (Fortran) used for exchanging data between processes. An MPI library exists on most, if not all, parallel computing platforms so it is highly portable.

MPI's send and receive calls operate in the following manner. First, process *A*decides a message needs to be sent to process *B*. Process A then packs up all of its necessary data into a buffer for process B. These buffers are often referred to as *envelopes* since the data is being packed into a single message before transmission (similar to how letters are packed into envelopes before transmission to the post office). After the data is packed into a buffer, the communication device (which is often a network) is responsible for routing the message to the proper location. The location of the message is defined by the process's rank.

Even though the message is routed to B, process B still has to acknowledge that it wants to receive A's data. Once it does this, the data has been transmitted. Process A is acknowledged that the data has been transmitted and may go back to work.

Sometimes there are cases when A might have to send many different types of messages to B. Instead of B having to go through extra measures to differentiate all these messages, MPI allows senders and receivers to also specify message IDs with the message (known as *tags*). When process B only requests a message with a certain tag number, messages with different tags will be buffered by the network until B is ready for them.

Difference between CORBA, RMI, Sockets, RPC.

Sockets seem easier to control, where something like CORBA could take a lot of time to learn. RMI would limit you client to needing to be another Java application. In case you don't know, RMI is a Java only version of CORBA.

Sockets would limit your client applications to also being Java based if you choose to write objects across the stream (So in answer to your question, yes you can serialize and send objects across a network)

sockets could be client independent (c++,.NET, etc) if you choose to just write bytes across the network. However writing bytes you would need to make some sort of protocol for communications, this can be a lot of work, and if you get errors and need to debug problems by reading byte streams... your in for a tough time.

CORBA is good for business. It allows you to read and write objects, and call methods on remote objects, and its highly optimized, its a mature technology, having been around well over 10 years. It allows your Java application to communicate with ANY other client, c++, .NET, etc

Difference between RMI & RPC

The main difference between RPC and RMI is that **RMI involves *objects***. Instead of calling procedures remotely by use of a proxy *function*, we instead use a proxy *object*.

There is greater transparency with RMI, namely due the exploitation of objects, references, inheritance, polymorphism, and exceptions as the technology is integrated into the language.

RMI is also more advanced than RPC, allowing for *dynamic invocation*, where interfaces can change at runtime, and *object adaption*, which provides an additional layer of abstraction.

Difference between CORBA & RMI

CORBA is a "language-neutral" distributed object scheme **-** this means you can have a C++ client and a Java server, and as long as they use CORBA to communicate, they can "talk" to each other. You basically create an interface you want the client and server to agree upon in IDL (CORBA "language" used to define "language independent" interfaces), and run an IDL compiler on it that produces corresponding interfaces for both languages and produces some "plumbing" code for them to talk to each other.

RMI is a more "Java only" distributed object scheme (though there are some projects that allow other languages to be able to "talk" RMI, it's pretty Java-centric). Basically, you make an interface extending

Java's "special" Remote interface, implement that interface in your server and run "rmic" on it to produce "plumbing" code (stubs and skeletons). If you want your RMI objects to be able to talk to CORBA objects you can use RMI-IIOP - IIOP is the protocol that CORBA objects use to "talk" to each other, so if RMI uses that to "talk" it is treated like any other CORBA object.

INPUT: A String.
for example (Message)
OUTPUT: Reversed String
(egasseM)
OPERATIONAL STEPS REQUIRED: ----

Following steps are required:

1. **Generate client and server code**
2. **compile client and server**
3. **Run server**
4. **Run client**

1. **Generate sample client and server code**

```
gedit server.c
        add code to it.

gedit client.c
        add code to it.
```

2. **Compile client and server**

```
mpicc server.c -o server
mpicc client.c -o client
```

3**. Run server**

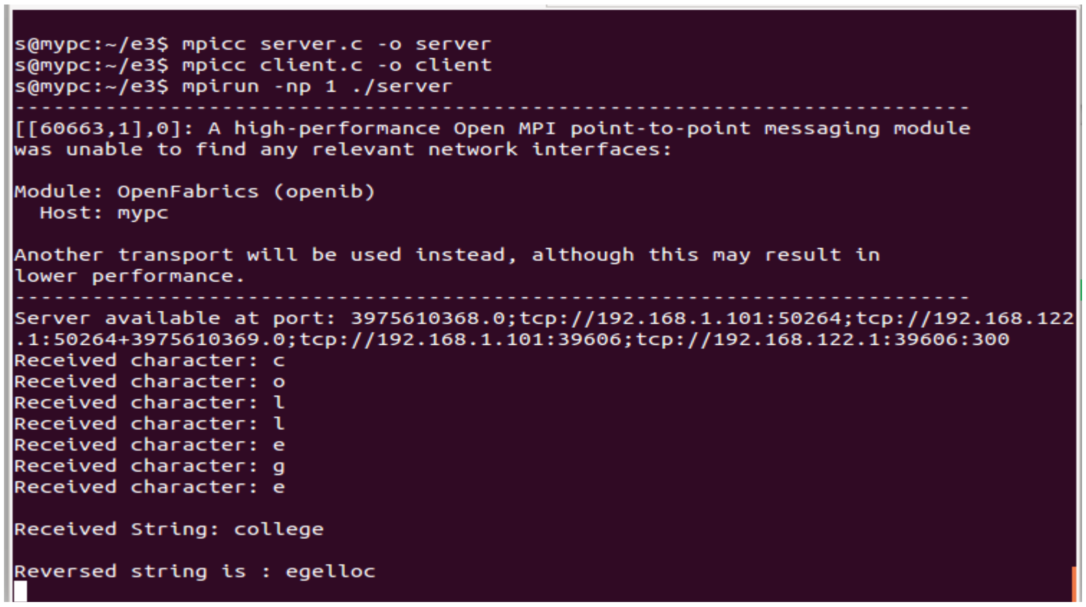
```
mpirun -np 1 ./server
# it will display output similar to below (not necessarily the same)
Server available at port:
4290510848.0;tcp://192.168.1.101:35820;tcp://192.168.122.1:35820+4290510849.0;tcp://192.16
8.1.101:40208;tcp://192.168.122.1:40208:300
```

# copy the **port-string** from the terminal output

4. **Run Client**

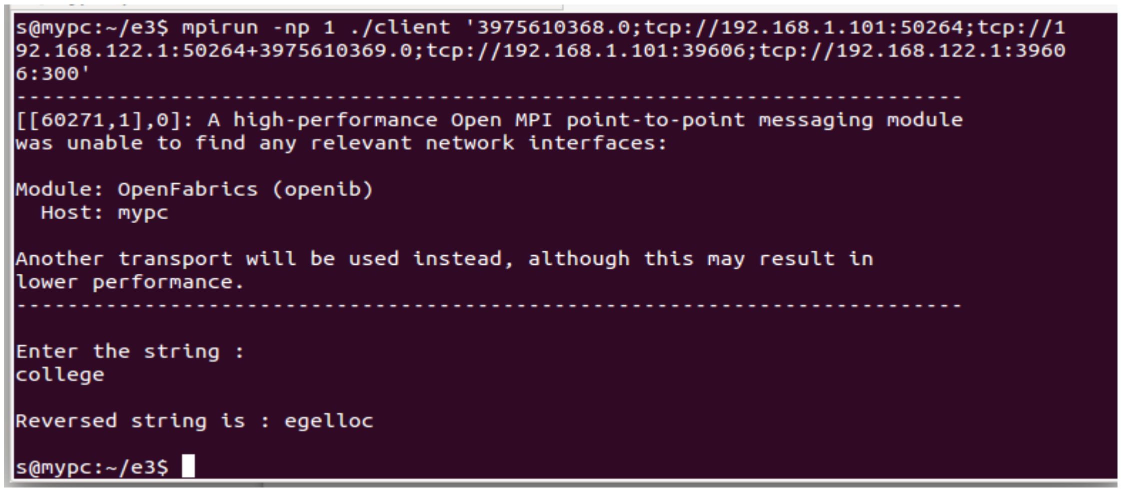
```
mpirun -np 1 ./client
'4290510848.0;tcp://192.168.1.101:35820;tcp://192.168.122.1:35820+4290510849.0;tcp://192.16
8.1.101:40208;tcp://192.168.122.1:40208:300'
```

Server



```
s@mypc:~/e3$ mpicc server.c -o server
s@mypc:~/e3$ mpicc client.c -o client
s@mypc:~/e3$ mpirun -np 1 ./server
--------------------------------------------------------------------------
[[60663,1],0]: A high-performance Open MPI point-to-point messaging module
was unable to find any relevant network interfaces:

Module: OpenFabrics (openib)
  Host: mypc

Another transport will be used instead, although this may result in
lower performance.
--------------------------------------------------------------------------
Server available at port: 3975610368.0;tcp://192.168.1.101:50264;tcp://192.168.122
.1:50264+3975610369.0;tcp://192.168.1.101:39606;tcp://192.168.122.1:39606:300
Received character: c
Received character: o
Received character: l
Received character: l
Received character: e
Received character: g
Received character: e

Received String: college

Reversed string is : egelloc
```

Client



```
s@mypc:~/e3$ mpirun -np 1 ./client '3975610368.0;tcp://192.168.1.101:50264;tcp://1
92.168.122.1:50264+3975610369.0;tcp://192.168.1.101:39606;tcp://192.168.122.1:3960
6:300'
--------------------------------------------------------------------------
[[60271,1],0]: A high-performance Open MPI point-to-point messaging module
was unable to find any relevant network interfaces:

Module: OpenFabrics (openib)
  Host: mypc

Another transport will be used instead, although this may result in
lower performance.
--------------------------------------------------------------------------

Enter the string :
college

Reversed string is : egelloc

s@mypc:~/e3$
```

ADVANTAGES and DISADVANTAGES OF MPI:

- Pros of MPI
  - runs on either shared or distributed memory architectures
  - can be used on a wider range of problems than OpenMP
  - each process has its own local variables
  - distributed memory computers are less expensive than large shared memory computers
- Cons of MPI
  - requires more programming changes to go from serial to parallel version
  - can be harder to debug
  - performance is limited by the communcation network between the nodes

APPLICATIONS OF MPI:

Tuning MPI Applications for Peak Performance

MPI is now widely accepted as a standard for message-passing parallel computing libraries. Both applications and important benchmarks are being ported from other message-passing libraries to MPI. In most cases it is possible to make a translation in a fairly straightforward way, preserving the semantics of the original program. On the other hand, MPI provides many opportunities for increasing the performance of parallel applications by the use of some of its more advanced features, and straightforward translations of existing programs might not take advantage of these features. New parallel applications are also being written in MPI, and an understanding of performance-critical issues for message-passing programs, along with an explanation of how to address these using MPI, can provide the application programmer with the ability to provide a greater percentage of the peak performance of the hardware to his application. This tutorial discusses performance-critical issues in message passing programs, explain how to examine the performance of an application using MPI-oriented tools, and show how the features of MPI can be used to attain peak application performance.

CONCLUSION:

It gives overall idea about middleware's like RPC, RMI, CORBA, MPI etc. which are used in the message transfer processes. It particularly focuses on Message Passing Interface(MPI),its working and related concepts.

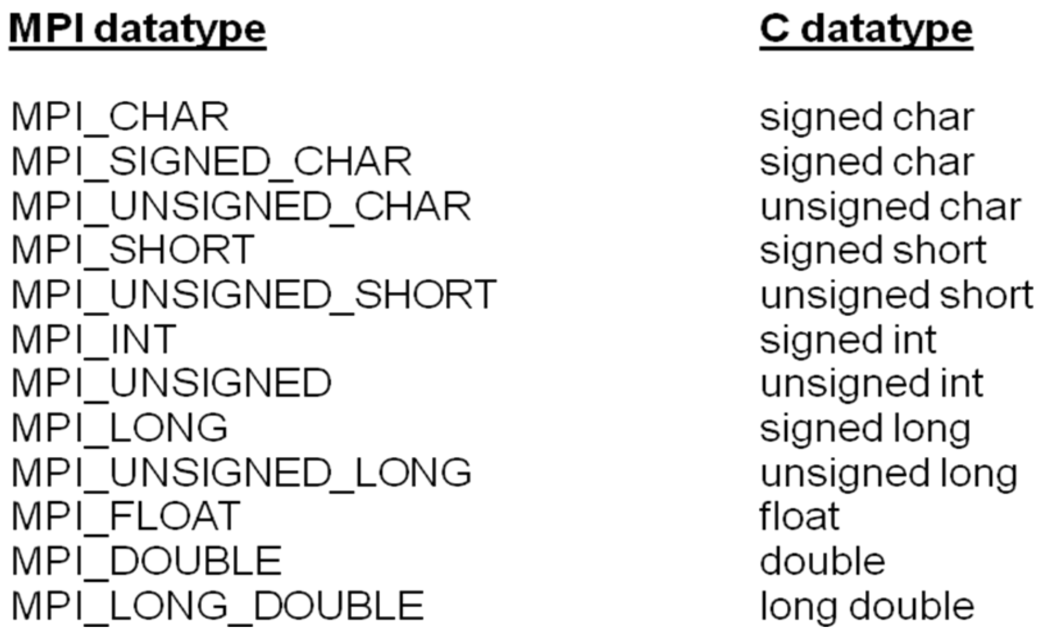
REFERENCES:

FAQ'S:

1) **What are the different data types of MPI?**

The data message which is sent or received is described by a triple (address, count, datatype).
The following data types are supported by MPI:

- Predefined data types that are corresponding to data types from the programming language.
- Arrays.
- Sub blocks of a matrix
- User defined data structure.
- A set of predefined data types



MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_SHORT	signed short
MPI_UNSIGNED_SHORT	unsigned short
MPI_INT	signed int
MPI_UNSIGNED	unsigned int
MPI_LONG	signed long
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

2) Explain various communication modes in MPI.

a. Synchronous: Completes once the acknowledgement is received by the sender.
b. Buffered send: completes immediately, unless if an error occurs.
c. Standard send: completes once the message has been sent, which may or may not imply that the message has arrived at its destination.
d. Ready send: completes immediately, if the receiver is ready for the message it will get it, otherwise the message is dropped silently.

3) Comment on basic commands of MPI.

The initialization routine MPI_INIT is the first MPI routine called.
MPI_INIT is called once

int mpi_Init(int *argc, char **argv);

The default communicator is the **MPI_COMM_WORLD**
A process is identified by its rank in the group associated with a communicator.
MPI_SEND(void *start, int count,MPI_DATATYPE datatype, int dest, int tag, MPI_COMM comm)
The message buffer is described by (start, count, datatype).
dest is the rank of the target process in the defined communicator.
tag is the message identification number.
MPI_RECV(void *start, int count, MPI_DATATYPE datatype, int source, int tag, MPI_COMM comm, MPI_STATUS *status)
Source is the rank of the sender in the communicator.
The receiver can specify a wildcard value for souce (MPI_ANY_SOURCE) and/or a wildcard value for tag (MPI_ANY_TAG), indicating that any source and/or tag are acceptable

Status is used for exrtra information about the received message if a wildcard receive mode is used.
If the count of the message received is less than or equal to that described by the MPI receive command, then the message is successfully received. Else it is considered as a buffer overflow error.

4) What is group communication?

The interaction of three or more interdependent members working to achieve a common goal. Group members use verbal and nonverbal messages to generate meanings and establish relationships.
Group communication requires interaction.
Defines and unifies a group
A clear, elevated goal:

- separates successful from unsuccessful groups
- guides action
- helps set standards
- helps resolve conflict
- motivates members

Each group member is affected and influenced by the actions of other members.
Group members work together to achieve a common goal.