Chapter 14

Distributed Programming Frameworks

Recall that distributed systems are characterized as a collection of autonomous computers communicating over a network of links. Some typical features of such a system are:

- There is no common physical clock and clocks can drift. Systems can be asynchronous too.
- The set of machines do not have common shared memory. This requires them to use messages for communication along with their semantics.
- The systems can be widely separated in a geographical sense. This allows them to be used in a variety of application such as SETI [3]. However, this comes with the challenge of programs having to deal with large message latencies.
- Nodes in the system are loosely coupled but cooperate with each other to achieve a common goal.
- Systems are naturally heterogeneous in their capabilities and functionalities. In such cases, it may not be possible to run some computation on some machines.

For the above reasons, distributed programming is inherently challenging. In essence, one needs to ensure that several geographically separate computers collaborate efficiently, reliably, transparently, and in a scalable manner. This idea leads to the notion of inherent complexity and accidental complexity as Brooks [20] mentions in the context of software systems.

Inherent Complexity: Inherent complexity refers to aspects of distributed programming that are inherent to the system model. Inherent complexity arises due to factors such as communication latency in the system, failures and recovery from failures of individual or groups of nodes in the system, and lack of physical clock which induces difficulties in arriving at an ordering of the events. Other factors that introduce inherent complexity in distributed programming include the possibility that failures in the system can introduce partitions that render unable communication across the system. Finally, there exist multiple aspects of security that the application has to contend with.

Accidental Complexity: Accidental complexity refers to aspects of distributed programming that can be addressed with appropriate technological interventions and enhancements. Accidental complexity arises due to using low level APIs that do not provide a good level of abstraction, lack of appropriate support for debugging and fault localization, and improper choice of algorithmic techniques. In addition, various components of distributed systems evolve in different trajectories and distributed programs have to contend with changes to key components in the life cycle of the distributed program.

14.1 Common Ideas Across Distributed Programming Frameworks

In this section, we try to highlight common features across most distributed programming frameworks.

14.1.1 Client-Server Vs. Peer Model

Distributed programming framework usually support two models. In the Client-Server model, the client application and the server program usually reside on independent machines and the programming framework allows for interaction between the client and the server and provides the necessary support for such interaction. Viewed differently, client request a certain service from the server and the server responds with the service. In the peer model, nodes participate as peers and there is no designated set of machines labeled as clients or servers.

Examples of the former include Remote Procedure Call (RPC), gRPC, GraphQL, Thrift, Kafka, and the like. In the latter, we have frameworks such as MPI, Erlang?, Map-Reduce?,

14.1.2 Synchronous Vs Asynchronous

One can also study distributed programming frameworks in terms of their support for synchronous communication only or also support for asynchronous communication. In synchronous communication, the sender and the receiver are waiting for the transmission and receipt of data. With asynchronous communication, the sender and the receiver do not have such a-priori arrangement. There is an additional level of detail with respect to message passing semantics. Some programming platforms support a blocking model where the sender an/or the receiver may be waiting for the send and the receive, respectively, to complete before moving to the next instruction in the program. In the non-blocking mode, the sender and receiver do not wait for the send and the receive to complete, respectively, before moving to the next instruction.

MPI offers support for both synchronous and asynchronous communication. More details of this are available in Section 14.2. In the synchronous case however, the programmer has to be careful to not introduce deadlocks in the code. (See Section **??** for more).

14.1.3 The Three P's

Just like parallel programming frameworks and languages, distributed programming frameworks also have to deal with striking a balance between the three P's of Portability, Performance, and Productivity. Portability refers to the ability of programs to work across different systems. Performance refers to the aspect of obtaining quality performance from a given program. Productivity refers to the ease with which programs can be expressed in the framework. Figure 14.1 shows the three P's along with distributed programming frameworks that satisfy these corners.

Distributed programming platforms such as MPI and gRPC are highly portable in general. On the other hand, platforms such as Map-Reduce offer high levels of productivity.

14.1.4 Blocking Vs Non-blocking Communication

There are two semantic models for communication between nodes in a distributed system. In the blocking mode of communication, the node that issues the receive command blocks itself from further processing until the

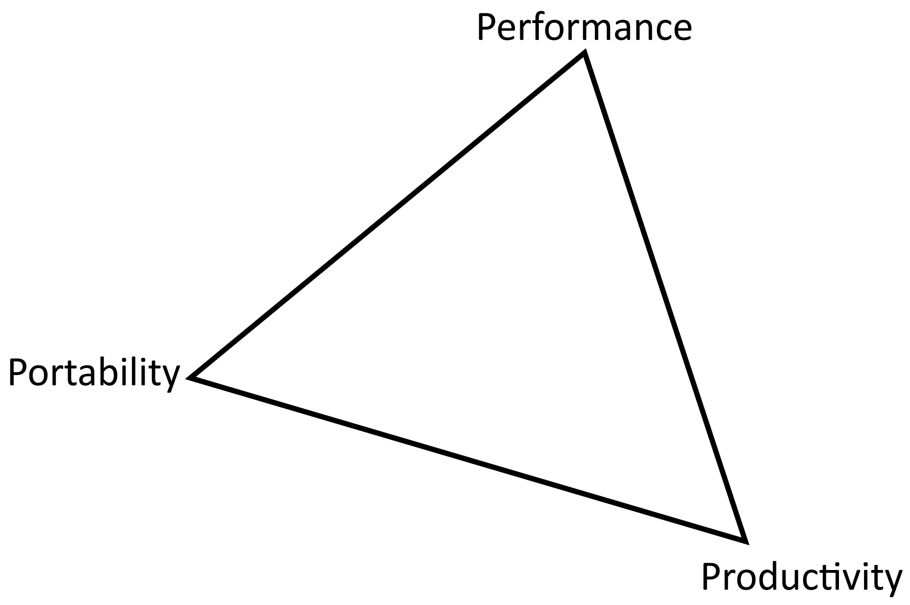

Figure 14.1: The three corners of distributed programming frameworks.

message is received from the intended receiver of the message. In the non-blocking mode, the node does not wait for the data transfer to actually happen. In particular, in a non-blocking receive operation, the node moves to the next instruction even if the data may not have arrived from the intended sender.

Both these modes have their particular use cases, especially in distributed programming. Various distributed programming frameworks support either or both of these communication modes. Indeed, it is possible to also talk about communication modes such as synchronous blocking, synchronous non-blocking, asynchronous blocking communication.

14.1.5 Common Primitives

Distributed programming frameworks usually support a set of common primitives for communication mechanisms such as send() and receive(), synchronization constructs such as wait(), and in some cases collective operations that work on data at every node.

To summarize, we show Table **??** that shows the support that various current distributed computing platforms provide.

While the three platforms we describe in this chapter are representative,

Characteristic	MPI	gRPC	Map-Reduce
Synchronous			
Asynchronous			
Blocking			
Non-blocking			
C-S			
P-P			

there exist other platforms too. Some of them such as Erlang are application specific, some such as Hadoop, GraphQL, are open-source versions of the above, and some such as CORBA, Java RMI, are earlier avatars of the ones we describe in the following.

14.2 MPI

Message Passing Interface (MPI) is a specification for a message passing library. This does not refer to a product or an implementation and is not specific to any compiler or programming language. The specification is defined as a standard and posted at www.mpi-forum.org. Since its definition, there have been numerous text books and other online material describing the MPI framework and its application to several problems. In this section, we provide a brief summary of MPI along with an example program.

MPI builds on the notion of a process and provides mechanisms for communication among processes running on parallel computers, clusters, or heterogeneous networks. MPI is a portable framework and is a useful way to provide for libraries that eventually obviates the need for every user to be an expert at writing MPI programs.

MPI combines interprocess communication with synchronization. In general, consider just two processes, Process P_0 and P_1. If P_0 sends some data intended for P_1, this is done by an explicit send operation. The receiving process, P_1, also blocks when expecting data from P_0. Processing at P_1 resumes only after receiving data from P_0. Note however that MPI-2 allows a divergence from this semantics and decouples communication and synchronization. These are called as one-sided operations.

MPI supports a peer model of interprocess communication. Each participating process gets a unique identifier denoted `rank`. A process can obtain its rank by using the MPI function `MPI_COMM_RANK()`. This function returns a number between 0 and the one less than the number of processes that identifies the rank of the process calling the function. Any participating process can know the number of participating processes via the MPI function `MPI_COMM_SIZE()`. This function returns the number of processes.

Using the above two functions, one can already write a simple MPI program as shown in Listing 14.1.

Listing 14.1: A first Simple MPI Program

```c
#include <stdio.h>
#include <mpi.h>
int main( int argc, char *argv[] )
{
    int myRank, total;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myRank );
    MPI_Comm_size( MPI_COMM_WORLD, &total);
    printf("My rank is %d out of %d processes.\n",
      rank, size );
    MPI_Finalize();
    return 0;
}
```

The program produces an output that has one print statement per each participating process. In the program above, the parameter MPI_COMM_WORLD refers to the default group to which all participating processes belong to. Processes can be organized into more groups based on application specific requirements. For instance, in a program that operates on a matrix input, all the processes that work on elements in a row can be made as part of a group. The function MPI_COMM_SPLIT creates a new group. To identify a new group, the MPI_COMM_SPLIT function takes in a parameter passed by reference and the value of this parameter is a way to identify the new group. As an example, suppose the default group MPI_COMM_WORLD has 16 processors and we want to create a subgroup that contains processors that have an even rank. The split command to achieve this is MPI_SPLIT(MPI_COMM_WORLD, rank/2, rank, &evenGroup) where rank contains the rank of the processor in the default group. This rank can be obtained using the MPI_Comm_rank function as in Listing 14.1.

In addition to groups, MPI uses another concept called the context. Messages are sent and received in a specific context. A context and a group together form a communicator. Each process participating in a group has a rank within that group. In the earlier program listed in Listing 14.1, the ranks are between 0 and one less than the number of processes since all processes belong to the default group MPI_COMM_WORLD.

In our discussion and examples in the following we will restrict ourselves to using a single group, the default group.

MPI Send

We move to describe the semantics and syntax of the MPI send function. The function MPI_SEND takes six parameters as input and has the following syntax.

```
MPI_SEND(start, count, datatype, destination, tag, group)
```

The first three parameters, start, count, and datatype, define the message buffer. The fourth parameter destination refers to the number of the receiving process in the communication group corresponding to the sixth parameter. The fifth parameter, tag, helps the receiver in identifying the message. This is useful in a context where one process sends multiple messages to a receiver and with possible message reordering, the receiver needs additional information to contextualize the received messages. In order to simplify the usage, MPI allows a default tag MPI_ANY_TAG.

Note that tags are different from data types. Data types allow for applications using MPI to specify the type of the data. This is useful when the machines on which the processes execute have varying formats and representations for various data types. Data types therefore help MPI support heterogeneity naturally.

MPI Receive

We now describe the semantics and syntax of the MPI receive operation. The receive operation, MPI_RECV takes seven parameters as input has the following syntax.

```
MPI_RECV(start, count, datatype, source, tag, comm, status)
```

A process executing the MPI_RECV function waits until a matching (on source and tag) message is received from the system and the buffer can be used. The match is applicable to both the specified source and the tag, the fourth and the fifth parameters of the function. The source is specified as the rank of the sender process in the communicator `comm`. If no specific sender is intended, then MPI_ANY_SOURCE is to be specified as the fourth parameter.

The message is copied to the buffer specified by the first three parameters start, count, and datatype. Receiving fewer than count occurrences of datatype is not an error but receiving more is an error.

The last parameter `status` contains information on the status of the receive operation.

Listing 14.2 show a small extension to Listing 14.1 by having the process with rank 0 to send a number to all the other processes. The other processes receive the number and print the number they receive.

Listing 14.2: Using MPI_Send and MPI_Recv

```c
#include <stdio.h>
#include <mpi.h>
int main( int argc, char *argv[] )
{
    int myRank, total, number;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myRank );
    MPI_Comm_size( MPI_COMM_WORLD, &total);
if (myRank == 0) {
    for (i=1;i<total;i++) {
        number = rand()%100;
        MPI_Send(&number, 1, MPI_INT, i, 0,
            MPI_COMM_WORLD);
    }
} else if (myRank != 0) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("Process %d received number %d from process 0\
        n",
            myRank, number);
   }
    MPI_Finalize();
    return 0;
}
```

14.2.1 Collectives

Recall that MPI structures send and receive functions to a group of processes. Collective communication is another support MPI offers to applications where the communication is now over all processes in a given group. There are three different types of collective operations that MPI supports. These are for synchronization, collective computation, and data movement.

Collectives for Synchronization

Applications sometime need to ensure that all processes within a group reach a specific point in execution before *any* of them proceeds further. The MPI_BARRIER function supports such a model. The syntax for this operation is `MPI_BARRIER(group)`, where `group` is a communication group.

Processes in the specified group block until *all* processes in the group also call the function. This requires processes to wait at this function and forces a synchronization of all the processes in the group. While the use of such barriers is seen as not being conducive to parallel and distributed programming, there are limited use cases where such barriers are useful.

Collectives for Data Movement

There exist several MPI functions that support movement of data over the process in a group. These come under the categories of one-to-all, all-to-one, and all-to-all communication.

In the first category, we have the broadcast and scatter operations. The broadcast operation, MPI_BCAST, has the following syntax and semantics.

```
MPI_BCAST(start, count, datatype, root, comm)
```

The first three parameters are as in the MPI_SEND function. The fourth parameter is the process that acts as the source of the broadcast. The message is sent to all processes in the group specified by `comm`.

Another operation that MPI supports in the first category is the scatter operation. The semantics of the scatter operation in general is to distribute a set of data items to a set of processes. The syntax of this function is as follows.

```
MPI_SCATTER(sData, sCount, sDatatype, rData, rCount,
    rDatatype,
              root, comm)
```

The first parameter indicates the source of the data to be sent. The second parameter specifies the number of data items of type sDatatype that is to be delivered to each of the processes in the communication group specified by the last parameter `comm`. The parameters prefixed with "r" refer to the buffer, number and datatype at the destination process. From sData, sCount data items of the specified type are moved to each of the processes in the group. The first sCount items go to the process with rank 1 (in the group), the next sCount items go to the process with rank 2 and so on. The parameter `root` indicates the rank of the source process.

We now move to the second category of collective data movement operations. The all-to-one operations in MPI include the gather operation. The gather operation is in a way has the opposite semantics to that of the scatter operation. Just as the scatter operation works to move elements from a designated source to all other processes in the group, the gather operation achieves the opposite effect. Data from all other processes in the group is moved to a designated source process. In addition, the operation orders elements according to the rank of the process from which they were received.

The syntax of the gather operation, MPI_GATHER, is as follows.

```
MPI_Gather(sData, sCount, sDatatype, rData, rCount,
   rDatatype,
          root, comm)
```

The first three parameters indicate the buffers at each process that contain the data items being transferred. The parameter `root` is the destination process. For this process, the parameters rData must be a valid buffer and the other processes can send a NULL filed for this parameter. Another detail that is important to note is that the field rCount is the number of data items received from each process and not the total number of data items received.

The final category of collective data communication operations include the all-to-all category. There are two operations in this category: MPI_ALLGATHER and MPI_ALLTOALL.

The first of these transfers data from each process the following

The second of these works in the following setting. Each process in a communication group has data. This data at each process is to be sent to all other processes. While this communication can be achieved by means of individual broadcasts, there is scope for an efficient implementation of the All-to-All communication especially when the data involved is small in size. The syntax of this function is as follows.

```
int MPI_Alltoall(sData, sCount, sDatatype, rData, rCount,
    recvDatatype, comm)
```

The first three parameters correspond to the data buffer, the number of data items, and the type of the data being transferred. The next three parameters prefix “r” correspond to similar ones at each process. The last parameter refers to the communication group.

14.2.2 Collectives for Computation

We now describe the collectives that perform a computation based on the data across the processes in a communication group. Prominent among these are the Scan and Reduce operations. These terms are popular in the parallel computing parlance and have the following meaning. A reduce operation applies an associative operation on data items from each process and produces one output. A scan operation applies an associative operation on data items from each process and produces one output at each process corresponding to applying the function data items at processes of ranks at most the process. REWORD.

The syntax for the MPI scan operation is as follows.

```
MPI_SCAN(sendbuf, recvbuf, int count, datatype,  op, comm
    )
```

In the MPI_SCAN function, the parameters sendbuf and recvbuf correspond to the input data items and the space for the output data item. The parameters count and datatype correspond to the number of data items and the type of the data. The parameter `op` refers to the operation to be applied. MPI has inbuilt support for standard operations such as addition, maximum/minimum, product, bitwise and/or, logical and/or, and the like. These are usually specified as MPI_SUM, MPI_MAX, and so on. The last parameter `comm` refers to the communication group on which the function is applied. As an example, if the group has four processes with values 2, -2, 1, and 4, and the operation is MPI_SUM, the result of the scan operation is 2, 0, 1, and 5.

The reduce operation, MPI_REDUCE has the following syntax.

```
MPI_REDUCE(sDdata, rData, count, datatype, op, root, comm
    )
```

The parameters have meanings similar to that of the MPI_SCAN function. One difference is that the result is stored in the process with rank specified by `root`. The other processes can send NULL for the parameter `rData`. The function is applied to the processes in the group specified by `comm`. The parameter `op` refers to the operation.

To summarize, in Table **??** we provide the output of the various collective operations in the setting where there are four processes holding a value each.

Process	P_1	P_2	P_3	P_4
Broadcast				
Initial Value	4	3	1	2
Received Value	3	3	3	3
Scatter, root = 2, sCount= 2				
Initial Value		[3, 8, 6, 2, 1, 5, 7, 4]		
Received Value	[3, 8]	[6, 2]	[1, 5]	[7, 4]
Gather, root = 1, rCount=1				
Initial Value	4	3	1	2
Received Value	[4, 3, 1, 2]			
AllGather, root = 1, rCount=1				
Initial Value	4	3	1	2
Received Value	[4, 3, 1, 2]			
All-to-All				
Initial Value	4	3	1	2
Received Value	[4, 3, 1, 2]	[4, 3, 1, 2]	[4, 3, 1, 2]	[4, 3, 1, 2]
Scan, op=+				
Initial Value	4	3	1	2
Received Value	4	7	8	10
Reduce, root=4, op = MAX				
Initial Value	4	3	1	2
Received Value				3

Table 14.1: Table shows an example of various MPI collective operations.

14.2.3 A Simple MPI Program

Armed with the simple toolkit of the MPI functions, we now proceed to develop a simple MPI program. We consider the computation of π using the idea that the number of points that fall inside a unit circle that is inscribed in a unit square approaches $\pi/4$. We develop this program by having each process do the random experiment of picking a set of points and counting how many of these points fall inside the unit circle. The overall average is then used to approximate the value of π.

In this direction, let N be the number of samples we use and n be the number of processes. Each process draws N/n samples and uses the variable ni to denote the number of points that fall inside the circle.

Listing 14.3 shows the entire program.

Listing 14.3: Computing π using MPI

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int done = 0, N, myid, n, ni, i, rc;
    double PI = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&n);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (!done)  {
        if (myid == 0) {
           printf("Enter the total number of samples
                      (N): (0 quits) ");
           scanf("%d",&n);
           }
        MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (N == 0) break;
        ni   = N / n;
        sum = 0.0; sumi = 0;
        for (samples = 0; samples < ni; samples++) {
            // draw a point in the unit square
            x = (double)rand()/RAND\_MAX;
            y = (double)rand()/RAND\_MAX;
            //check if the point lies inside
            //the unit circle
            if (x*x + y*y <= 1) sumi ++;

        }
        pii = 4*sumi/ni;
        MPI_Reduce(&pii, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD);
         if (myid == 0)
         printf("pi is approximately %.16f, Error is %.16
            f\n", (double)pi/n, fabs(pi - PI25DT));}
         MPI_Finalize();
        return 0;
}
```

Building and Running the Program

Unlike standalone programs, the steps to follow to be able to run an MPI programs on a cluster is usually somewhat dependent on the installation. If you are running MPI on your multi-core machine, it is a bit straight-forward. We will only mention the steps needed to run an MPI program on a single multi-core machine. For the other case, it is best to discuss with your local cluster/facility administrator.

On your multicore machine/laptop, you need to first install the MPI package such as mpich or OpenMPI. This installation procedure depends on the OS that your machine runs on. For typical Linux based machines, you can use apt-get or yum to do the job. After this installation is complete, you use `mpicc` to compile MPI programs. You can use simple `make` scripts to set up a smooth compilation workflow. Once the executable is ready, you use `mpirun` to run the MPI program. The syntax is `mpirun -np <num> <file>`. In the above, <num> refers to the number of cores you want your MPI program to use and <file> is the name of the executable file.

Some Pitfalls

MPI supports synchronous and blocking send and receive operations. While most programs expect such a model, it is possible that deadlocks can be created due to these operations. Consider two MPI processes, P_1 and P_2, with P_1 waiting to receive data from P_2 post which it sends data to P_2, while at the same time, P_2 wants to receive data from P_1 before it sends data to P_1. Listing **??** shows the corresponding code snippets.

The trouble with the above listing is that P_1 waits to receive a message from P_2 and vice-versa. There exist several ways to avoid this and other such pitfalls. In this case, one can reorder the send and receive instructions

```
$\vdots$
MPI\_recv(2, &data, )
if (true) MPI\_send(2,
     reply, )
$\vdots$
```

```
$\vdots$
MPI\_recv(1, reply, )
if (reply) MPI\_send
    (1, &data, )
$\vdots$
```

at either P_1 or P_2 to remove the circular waiting. Another way to avoid this particular danger is to use asynchronous send and receive operations supported by MPI2. These are the iSend() and iRecv() operations and we leave it to the reader to explore the semantics of the asynchronous operations.

14.2.4 MPI Summary

MPI has been found to be quite useful in writing distributed programs and has a strong user base. There are some pitfalls one has to be aware of. In particular, the send and receive operations in MPI are blocking in their semantics. This can cause deadlocks. For instance, if the receiving process does not have enough space, then the sender waits for this space to be made available.

MPI2 offers support for non-blocking variants of the send and receive operations. These can avoid deadlocks but introduce other issues that the programmer has to be aware of.

MPI is a useful tool to create libraries, however one drawback is that MPI does not offer fault tolerance.

14.3 Map-Reduce

As we note from Section 14.2, MPI does not offer any fault tolerance. One of the possible explanations for this lack of support is that MPI often is used on server scale infrastructure that comes with fault tolerance. However, maintaining server scale infrastructure is often expensive. On the other hand, commodity hardware is ubiquitous and cheap but comes at the expense of reliability. One challenge therefore is to see if large-scale distributed programming can be supported on commodity hardware with additional support for fault tolerance.

Another lacunae of MPI is the lack of emphasis on productivity. MPI programmers have to deal with a low level of programming abstraction. Given the complexity and usage patterns of distributed programming, a high level of abstraction can help programmers achieve productivity without impacting performance.

These considerations led to the development of the distributed programming framework, Map-Reduce, by Dean et al. in 2005 [34]. The Map-Reduce framework runs on commodity hardware that are usually available in bulk and are less expensive. The framework has to however work with failures to machines or components of machines and also contend with low network and disk bandwidth.

Ever since the release of Map-Reduce, there have been several similar frameworks. Apache released Hadoop , which is an open-source version of Map-Reduce. Amazon has its own similar offering called PaaS, standing for Platform-as-a-Service. In the following, we explain the Map-Reduce programming framework along with a complete example.

14.3.1 The Map-Reduce Programming Model

The Map-Reduce programming model envisages expressing distributed programs via two functions: a mapper and a reducer. It is possible to express the solution to many problems from domains such as image processing, graph computations, and machine-learning algorithms as a sequence of mapper and reducer functions. While the model is not general-purpose and cannot express all computations, the versatility of the map-reduce model is in its ability to express several computations in the map-reduce model. REWORD–NOT SMOOTH TEXT

The Map Function The map function takes as input a list of key, value pairs. The output of the function is a list of key, value pairs. The map function produces zero or more values per each input key. The Map-Reduce framework supports certain in-built functions that can be applied during the mapper. Examples of such functions include toUpper() that converts a string argument to a string consisting of all uppercase letters.

The Reduce Function The Reduce function

A Simple Example Let us imagine that the Aadhar data is stored as a sequence of records, one per each line, with various fields such as Aadhar number, address pin code, first name, last name, year of birth, address, last update date, among other fields. Figure **??** shows an example set of records.

Suppose we want to find the most common first name by pin code. We treat each line of the input as a key,value pair and use the mapper function

1, 1093 6233 8742, 100205, vivek, rao, 1997, Apt 23 Krishna Layout Chennai, 2021-08-23-161523
2, 8372 7362 6532, 203303, sara, jacob, 2011, 1-3-231/A Roy Enclave Kolkata, 2023-05-16-100245
3, 3293 7462 8730, 650237, srinivas, krishna, 1986, House no 236 Gandhi colony Vijayawada, 2015-04-22-1435
⋮

⟨132980, [rama, vikas, rama, sai]⟩
⟨225682, [madhu, ankit, krishna]⟩
⟨763210, [vivek, sara]⟩
⟨132980, [kapil, charan, rama]⟩
⋮

to emit output key,value pairs of the form ⟨ pin code, first name ⟩. A sample set of such pairs look as follows.

⟨132980, rama⟩⟨225682, madhu⟩⟨763210, sara⟩⟨132980, rama⟩⋮

Now, the reduce function gets as input key-value pairs that have a list of values per each key. The key is the pin code and the list of values contains all the first names of people in that pin code as Figure **??** shows.

The reduce function can pick the most frequent name per each pin code and produce tuples of form ⟨pin code, frequent-first-name⟩.

Shuffle and Sort To understand how the output from a mapper is transformed as input to the reducer, we need to delve a little deeper into the Map-Reduce framework. In addition to the map and the reduce function, which are user-defined, the real support that the Map-Reduce framework provides is via its in-built functions that shuffle the output of mapper functions and the sorting of keys done before calling the Reducer function.

The shuffle operation applies to the key-value pairs from all mapper machines. The key-value pairs are arranged into groups based on the keys. In other words, all the key-value pairs with the same key are grouped together. The framework applies these functions once *all* the mappers complete their task.

Before sending the (grouped) key-value pairs to reducer machines, the Map-Reduce platform does a sort of the keys. Sorting ensures that the keys at any reducer machine arrive in a sorted order. There is no sorting guarantee of keys across reducer tasks.

CHECK: what is sorted? Keys or values within a Key?

The Partitioner and Combiner From the above paragraphs, it appears that a program written in the Map-Reduce framework has four phases: mapper-shuffle-sort-reducer. Of these, the user has to implement the Mapper and the Reducer functions while the framework has efficient support for the shuffle and sort functions. There are two other optional functions that the framework provides. These optional functions provide further opportunities to optimize the program.

The combiner function applies to the output of each mapper. The combiner runs on the mapper machines and the key,value pairs output and applies a reduce function on these key,value pairs. The framework however may call the combiner function zero, one, or more than one time. This means that the functions that the programmer can use as part of the combiner routine should be such that the output is unaffected by its repeated application. The real benefit of the combiner is to act as a mini-reducer thereby decreasing the volume of data that is carried over to the shuffle and the sort stages. However, the combiner cannot replace the role of the Reducer since the combiner function is applied only on the data local to the mapper.

The partitioner function allows the programmer a limited control on the reducer machine that key,value pairs from the shuffle and the sort stage go to. By default, the framework sends each key,value pair to a reducer machine chosen uniformly at random based on the key. This default partitioner, also called as the HashPartitioner, uses a hash function on keys to map each key to a reducer. The hash function that the framework uses is chosen so as to balance the distribution of keys to the reducer machines. Notice that balancing with respect to the number of keys is no guarantee on the number of values processed by a reducer.

The Map-Reduce Nomenclature

Datanode

namenode

14.3.2 A Complete Example: Histogram

Listing 14.4: Compuing a histogram using Map-Reduce

```
int bucketWidth = 8 // input

Map(k, v) {
    emit(floor(v/bucketWidth), 1)
    // <bucketID, 1>
}



// one reduce per bucketID
Reduce(k, v[]){
    sum=0;
    foreach(n in v[])  sum++;
    emit(k, sum)
    // <bucketID, frequency>
}
```

Building and Running the Program Based on the installation, there are minor challenges in how one can build and execute the program.

14.3.3 Other Features

Recall that Map-Reduce offers fault-tolerance so as to accommodate potential failures to machines or components of machines. Map-Reduce scales to run on thousands or more of commodity machines and hence failures are common. One way of providing for fault-tolerance is via the JobTracker and the TaskTracker. Note that while retaining the functionality, the second version of Map-Reduce calls these as ResourceManager and ApplicationMaster.

The JobTracker receives requests for executing MapReduce programs from the client. The JobTracker works in coordination with a TaskTracker in identifying and addressing failures. JobTracker assigns each MapReduce job to a TaskTracker based on parameters such as locality of the input data.

14.3.4 Other Variants

Map-Reduce is a product that came out of Google [34]. Recall that the Map-Reduce framework runs using the Google File System (GFS) as the underlying data store. Apache released an open-source version of Map-Reduce, Hadoop [4]. Hadoop runs using the Hadoop Distributed File System (HDFS), an Apache open-source variant of GFS, as the underlying data store.

YARN, standing for Yet Another Resource Negotiator, is another variant of Map-Reduce. In fact, YARN addresses some of the scalability concerns with Map-Reduce and can scale up beyond 10000 nodes. To this end, YARN has more sophisticated fault tolerance mechanisms. Recall from Section **??** the failure of the JobTracker endangers the Map-Reduce applications currently under execution. The ResourceManager of YARN handles this failure by saving enough state information.

14.3.5 The Map-Reduce and Other Related Technologies

We now discuss how the Map-Reduce platform differs from other related technologies.

Scripting Languages such as `awk` and `grep` Some of the mapper functionality and the reducer functionality can be implemented using sophisticated Unix style scripting languages such as `awk` and `grep`. Other high-level abstractions such as PERL and Python also provide a strong suite of libraries and routines to support typical mapper and reducer functionality.

However, the true power of Map-Reduce is in being able to offer an abstraction for parallel processing of large data. Tools such as `awk` and `grep` do not offer support for parallel processing. High level languages such as Perl and Python offer linkages to parallel and distributed programming frameworks but do not cater to the kind of abstraction that Map-Reduce can provide in particular for parallel processing.

Map-Reduce Vs. RDBMS It appears that the functionality that Map-Reduce supports can be achieved via relational database systems and their query languages. However, there are key difference between these two technologies.

- The scale of the data that Map-Reduce applies to can be an order of magnitude bigger than what a typical relational database can store. Map-Reduce is indeed aimed at processing big data.
- It appears that Map-Reduce would scan each record and apply a user function on the record. This can be supported easily by a query to

an RDBMS also. However, the read model of RDBMS is usually optimized for point queries. RDBMS also have a huge read latency and some of this latency is offset by using indexing. There is no such need to support indexing on data that is provided as input to the Map-Reduce framework.

- RDBMS works well with structured data. Most RDBMS table designs also perform normalization whose main goal is to remove redundant storage and provide for data integrity. One drawback of normalization is that a complete record read has to be translated to a set of non-local reads. On the other hand, Map-Reduce assumes that it is possible to perform high-speed streaming reads and writes.

- An RDBMS is good for point queries or updates, where the dataset has been indexed to deliver low-latency retrieval and update times of a relatively small amount of data.

However, the two technologies namely RDBMS and Map-Reduce are seeing a convergence of sorts. There exist RDBMS systems such as Aster Data and Greenplum that incorporate certain aspects of Map-Reduce in their support. Similarly, language abstractions such as those supported by Pig and Hive built on top of Map-Reduce frameworks offer support for SQL style operations for users.

14.3.6 Summary

To summarize, the Map-Reduce framework is versatile to support a variety of computations involving large-scale data. The framework provides a high degree of abstraction and offers the user a transparent mechanism to deal with many issues of distributed programming such as scheduling and load balancing, data and task distribution, synchronization, handling faults, and so on. On the other hand, the programmer has to express the computation via only four operations: map, reduce, combiner, and partitioner.

14.4 gRPC

Remote Procedure Call (RPC) is a popular interprocess communication mechanism that has been in vogue for several decades. Nelson [17] introduced the earliest version of RPC in 1976. SUN put its own version of RPC in 1980 and used it its implementation of NFS [112]. RPC provides an abstraction of allowing (client) programs to make function calls that get executed on a remote machine (server). The abstraction makes it transparent for the user to issue the function call. RPC itself internally uses an existing transport layer protocol such as UDP or TCP to prepare the function call, contact the server for executing the function, and report the results back to the client program.

Later versions of platforms that supported RPC included the middleware platforms such as Common Object Request Broker Architecture (CORBA) and Java RMI. These platforms added an object-oriented model to RPC and provided for data encapsulation and a separation of the method (function) interface and implementation. Google introduced gRPC [54] in the year 2015 and is based out of its own in-house RPC framework Stubby that allowed multiple microservices within the Google data centers to connect and communicate with each other. One trouble with Stubby is its strong coupling to Google internal microservices thereby rendering it less generic. gRPC is a open source framework that is general-purpose and cross-platform ready while offering similar scalability, performance, and functionality to that of Stubby.

Many practices from the original definition of RPC and its later versions such as CORBA [57], SOAP [19], and REST [41] continue to be used in later versions such as gRPC. Some of these include the ideas about the Interface Definition Language (IDL), marshaling and demarshaling of parameters, synchronous communication based request-response model, platform neutrality, and the like.

gRPC offers several advantages over other RPC platforms such as CORBA. These are mentioned in the following.

- Use of HTTP/2: Unlike CORBA and Apache Thrift, gRPC uses HTTP/2 for its communication. HTTP/2 has inherent advantages such as using a single network connection and increases the efficiency of inter-process communication.

- Strong Typing: gRPC uses strong static typing thereby reducing the scope of errors in distributed programming.

- Multi-language Support: gRPC offers support for multiple languages to be used at both the server and the client end.

- Full Duplex Communication: gRPC provides in-built support for full duplex communication and hence naturally extends support for developing streaming server and streaming client based applications.

This communication model extends the simple synchronous request-response style of communication that existing RPC platforms are limited to.

- Additional Features: gRPC has native support for additional features such as authentication, resiliency, data compression, load balancing, and the like.

14.4.1 Using gRPC

In this section, we outline the steps needed to write a gRPC based client-server application. We imagine a hypothetical set up where a university offers a service based on the courses offered in an academic year such as the list of courses. Clients can request information about a particular course or a set of courses. We will extend this scenario with other services and mechanisms such as streaming, transcripts, and the like.

The gRPC Server

We first identify what the server will post as the functions that become part of the service provided by the server. In our example, the university service aims to support functions such as getting the course name, course syllabus, number of seats available, and the meeting schedule of a given course code. The course code is a seven character alpha-numeric string such as CS2.312. The course name and the course syllabus are of type string, and the number of seats available is a positive integer. We now write these functions in the Interface Definition Language.

```

rpc getCourseName(String courseCode)

rpc getCourseSyllabus(String courseCode)

rpc getCourseSeatsAvailable(String courseCode)
```

The gRPC Client

In the following, we show the client program that uses the above functions.

Building the Application

Listing 14.5: The Protocol Definition

```
syntax = "proto3";

option java_package = "ex.grpc";

package CourseInfo;

// The course information service definition.
service CourseName {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

The client program looks as shown in Listing **??**. THIS IS A C++ PROGRAM. TO BE CHANGED TO JAVA

Listing 14.6: gRPC Client Example

```
#include <iostream>
#include <memory>
#include <string>
#include <grpc++/grpc++.h>
#include "helloworld.grpc.pb.h"
using grpc::Channel;
using grpc::ClientContext;
using grpc::Status;
using helloworld::HelloRequest;
using helloworld::HelloReply;
using helloworld::Greeter;
class GreeterClient {
 public:
  GreeterClient(std::shared_ptr<Channel> channel)
      : stub_(Greeter::NewStub(channel)) {}
  // Assambles the client's payload, sends it and
      presents the response back
  // from the server.
  std::string SayHello(const std::string& user) {
    // Data we are sending to the server.
    HelloRequest request;
    request.set_name(user);
    // Container for the data we expect from the server.
    HelloReply reply;
    // Context for the client. It could be used to convey extra information to
    // the server and/or tweak certain RPC behaviors.
    ClientContext context;
    // The actual RPC.
    Status status = stub_->SayHello(&context, request, &reply);
    // Act upon its status.
    if (status.ok()) {
      return reply.message();
    } else {
      std::cout << status.error_code() << ": " << status.error_message()
                << std::endl;
      return "RPC failed";
    }
  }
 private:
  std::unique_ptr<Greeter::Stub> stub_;
};
int main(int argc, char** argv) {
  // Instantiate the client. It requires a channel, out of which the actual RPCs
  // are created. This channel models a connection to an endpoint (in this case,
  // localhost at port 50051). We indicate that the channel isn't authenticated
  // (use of InsecureChannelCredentials()).
  GreeterClient greeter(grpc::CreateChannel(
      "localhost:50051", grpc::InsecureChannelCredentials()));
  std::string user("world");
  std::string reply = greeter.SayHello(user);
  std::cout << "Greeter received: " << reply << std::endl;
  return 0;
}
```

The server program looks as shown in Listing 14.7

Listing 14.7: gRPC Server Example

```cpp
void RunServer() {
  std::string server_address("0.0.0.0:50051");
  GreeterServiceImpl service;
  ServerBuilder builder;
  // Listen on the given address without any authentication mechanism.
  builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());
  // Register "service" as the instance through which we'll communicate with
  // clients. In this case it corresponds to an *synchronous* service.
  builder.RegisterService(&service);
  // Finally assemble the server.
  std::unique_ptr<Server> server(builder.BuildAndStart());
  std::cout << "Server listening on " << server_address << std::endl;
  // Wait for the server to shutdown. Note that some other thread must be
  // responsible for shutting down the server for this call to ever return.
  server->Wait();
}
int main(int argc, char** argv) {
  RunServer();
  return 0;
}
```

Installing and Executing a gRPC Program

Setting up gRPC requires a prior installation of the necessary compilers and other tools. One way to get this set up done on Linux based systems is to clone from the existing `git` repository at `https://github.com/grpc/grpc` and executing the `make` followed by `make install` command that runs the setup scripts.

Once the installation is available,

Adding Streaming Support

Support for streaming client-server interaction is one feature that gRPC provides. In the streaming communication model, for one request from the client, the RPC server can send multiple responses. These responses need not all be sent at once and can be sent as the server obtains more data that matches the response. The server marks the end of the stream by sending the status details of the server and trailing metadata to the client.

In terms of programming, the stream server model introduces a few differences. The return type of the server function that outputs a stream has to be given the type as a stream variable. At the client side, the program that calls such a stream output based remote service function has to process the multiple outputs until the end of the stream is reached.

In the gRPC framework, also the RPC client can operated in a streaming model. In this model, the client sends a sequence requests to the server. The server makes a single response to the entire sequence of requests. The server is not under an obligation to wait for all the requests to be received before sending its response. The server can send its response after reading one or more of the messages of the request sequence. The client program however includes a end of stream message to notify the server.

Applications that benefit from streaming request include those that involve large data transfers. The entire data can be chunked into more manageable sizes at the sending side. The receiving side receives chunks of the data. One benefit in such cases it to reuse an established connection for transferring all the chunks instead of establishing new connections for each chunk. Another such use case is where the server may be interested in sending a reply based on data that is yet to arrive or yet to be processed. For instance, consider suggesting friends to a user on the Facebook website. An initial set of, say a dozen, friends may be recommended at first. If the user continues to browse that functionality, the server may send another dozen recommendations. This allows the server to compute and send only when the response is in demand. In this use case, it is not the size of the data that warrants a proper use of streaming, but it is the cost associated with preparing the reply that the server optimizes on.

gRPC also allows applications to use bidirectional streaming where both the client and the server send a sequence of messages using a read-write stream. The two streams operate independently, so clients and servers can read and write in whatever order they like. For instance, the server could wait to receive all the client messages before writing its responses, or it could alternately read a message then write a message, or some other combination of reads and writes. The order of messages in each stream is preserved.

Using Authentication

Like RPC, gRPC also provides optional features to enable authentication of the server, the client, and the communication channel between the client and the server. Transport Level Security (TLS) supports the communication channel security in gRPC. A secure channel between the client and the server should offer support for a reliable and private connection between the client and the server. Reliability of the channel refers to the ability of the channel to support message integrity checking and to prevent alteration of data in transit in the channel. Private connection refers to the ability to use appropriate cryptographic mechanisms that establish a unique key for each connection. This unique key is usually set up via negotiation at the start of the connection.

gRPC supports a one-way or two-way secured connection. In the one-way mode, only the client validates the server to ensure that the messages received are indeed from the intended server. During connection establishment, the server shares its public certificate with the client. The client can verify the shared public certificate using a Certificate Authority (CA). Once the verification with the CA is successful, the client uses a shared key to send encrypted messages.

In the two-way mode, both the client and the server authenticate each other. Both the client and the server share their public certificate with each other. In turn, they approach the CA to validate the received certificate. Post the validation of the certificate by the CA, the client and the server use secure communication to continue their exchange. The client initiates the flow of these events by contacting the server, obtaining the public certificate of the server, verifying the certificate with a CA, sending the certificate of the client to the server post-verification, the server verifying the certificate of the client using a CA, and then enabling exchange between the client and the server.

Beyond a secure channel, gRPC also provides for authenticating the identity of a client at the server side. gRPC supports three possible mechanisms to this end: the client can provide a username:password in Base64 encoded form, the client can use JSON tokens, and the client can use OAuth2 tokens. One advantage in using tokens is that tokes can be set with a time duration after which they expire.

Other Functionalities

gRPC provides support for lots of other functionalities. One such is interceptors. These are pieces of code that get invoked prior to the RPC call and act as useful mechanisms to perform any pre- or post-processing as part of the RPC call. Some functionality that one can push to such interceptor routines include logging, metrics, authentication, and the like. Request interceptor is an interceptor that gets invoked during the request call and Response interceptor is an interceptor that gets invoked during the response call. Based on the nature of communication mechanism used by the client and the server, the program needs to implement and register the unary interceptor or the stream interceptor. gRPC allows for more than one interceptor function to be invoked as part of the RPC call. Example below shows the syntax of how to use an interceptor in the gRPC framework.

Other gRPC features provisioning for deadlocks, timeouts, cancellations, and error handling. For a complete list of gRPC features and supported functionality, we refer the reader to the book by Indrasiri and Kuruppu [67].

14.5 Others

Beyond the distributed programming frameworks detailed in the above sections, specific application domains benefited from application specific programming frameworks. Some such include Erlang [5], Emerald, Argus, and Orleans, that are based on message passing. There are also distributed programming languages based on the distributed shared memory model. Some of these include Mirage, Orca, and Linda. For a brief introduction to these variants, we refer the reader to [97].

There are other domain specific distributed programming frameworks such as Pregel [91] and GraphX [118] for graph processing, Graphlab [87] and TensorFlow [2] for machine learning algorithms, and the like. These frameworks allow programmers to write distributed programs for specific domains to express their program as a sequence of high-level steps that are supported by these frameworks. This results in higher productivity and also higher performance as the high-level steps are usually well-optimized.

14.6 Summary

There exist multiple languages and frameworks for programming distributed systems. This chapter did not discuss issues such as best practices in distributed programming, the idea of microservices, software architectures for distributed programming, fault location and debugging, and many other associated issues. For instance, writing efficient distributed programs requires good understanding of problem decomposition, studying the impact of interprocess communication vis-a-vis local computation, process synchronization, and the like. The book by Quinn [109] discusses these techniques in detail.

In addition, debugging distributed programs is often challenging since it may be difficult to replicate a scenario leading to a fault as events run across multiple processors.

14.7 Questions

1. Understand the difference between HTTP/1.0 and HTTP/2.0 by preparing web pages that use the two protocols and the time taken to load web pages using these two protocols.

2. Study the benefits of client-side, server-side, and bidirectional streaming in gRPC using suitable examples. As part of this experiment, you can study the transfer time and the connection establishment time and hence understand the savings introduced by using the streaming model.

3. Discuss why a platform such as MPI does not provide any support for streaming connections like gRPC.

4. Think of problems that are very suitable to solve with each of MPI, Map-Reduce, and gRPC, and discuss why. Similarly, identify problems that are most unsuitable for the above frameworks and discuss why.