Distributed programming with service-oriented languages

IADM816: ISA in Computer Science

Julie Boysen

IMADA
University of Southeren Denmark
Supervisor: Marco Peressotti
January 30, 2023

Contents

1 Introduction

In this ISA one of the challenges of distributed programming is tackled, namely managing communication among multiple nodes in a distributed system. Group communication algorithms are used to ensure that all nodes in a group receive the same information. In this ISA the group communication algorithm implemented is the multiple sources single group (MSMG) algorithm, with ensured causal ordering of messages and using broadcast as the method of message cast.

The goal of this ISA is to implement group communication as a library in the programming language Jolie. This will allow Jolie services to manage communication among multiple nodes in a distributed system.

2 Problem description

In this ISA the multiple sources single group (MSMG) group communication algorithm is implemented, with ensured causal ordering of messages and using broadcast as the method of message cast. This algorithm allows multiple sources to send messages to a single group and ensures that the messages are received by all members of the group in the order they were sent.
To accomplish this, a logical clock is needed, and in this case vector time is used as the logical clock to track the progress of each node in the group and ensure that messages are broadcast and received in a causally consistent manner.

The vector time of a node i is represented by vt_i, where $vt_i[i]$ is the node's local clock and $vt_i[k]$ shows node i's view of node k's logical local time. The clock updates according to these two rules[1]:

- Before performing an event on node i, the logical clock is incremented by a fixed value d (usually 1), $vt_i[i] = vt_i[i] + d$.
- Each message carries the sender's logical clock value at the time of sending. When node i receives a message with timestamp vt_m, it updates its own clock value to the maximum of its current value based on the timestamp of the received message, $vt_i[k] = max(vt_i[k], vt_m[k])\forall k$, and applies rule 1. The message is then delivered to the application.

For group communication, when a node sends a message the first rule of the clock is applied. Upon receiving a message from another node, it is placed in the delivery queue and the second rule of the clock is applied. To deliver messages to other nodes, all messages with timestamps less than or equal to the current value of the receiving node's clock are broadcast to all other nodes.

3 Architecture

To gain an understanding of the general concept behind the implementation of the group communication library in Jolie, Figure 1 is provided. Figure 1 depicts an example of how services within a distributed system using the group communication service can interact with one another. There are three services, each with an attached group communication service.

For this ISA, a basic server architecture has been implemented, in Figure 1 there are depicted 3 servers `S1`, `S2`, and `S3`. As seen in Figure 1, the servers are only able to communicate with their

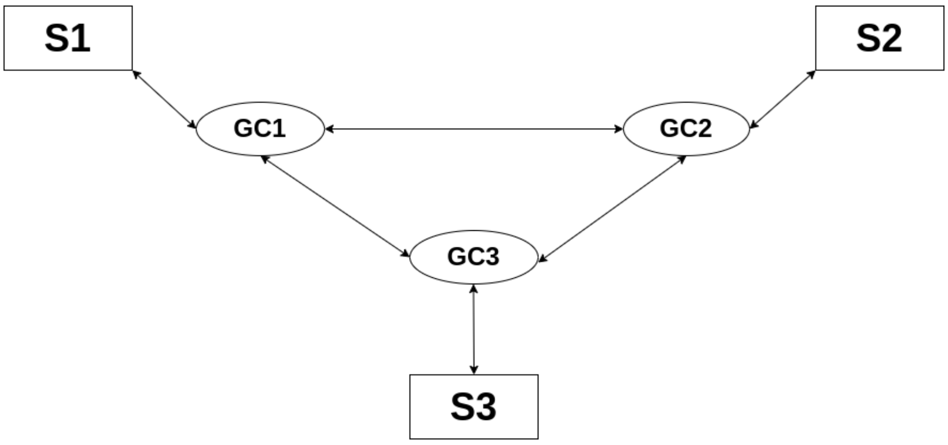

Figure 1: Example of how the group communication service would work in the case of 3 services needing to communicate

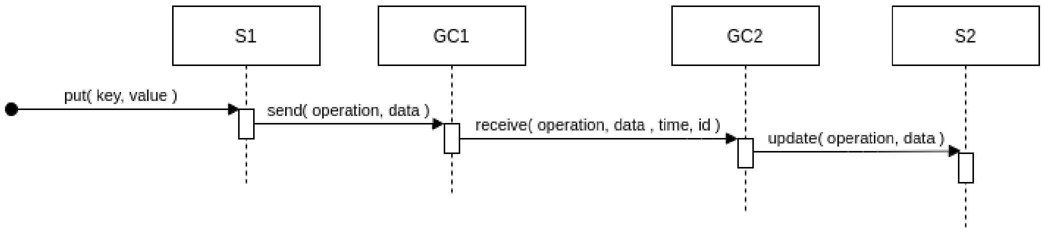
S1
GC1
GC2
S2
put(key, value)
send(operation, data)
receive(operation, data , time, id)
update(operation, data)

Figure 2: Example of how services communicate in the case of a put request

respective group communication service as it is responsible for maintaining the logical time by keeping track of all messages sent and received by the server and based on that broadcast messages to the other group communication services.

In Figure 2 an example of the sequence of function calls are illustrated. The figure illustrates an example of the function `put` called by an client on `S1`. `S1` then calls the function `send`, which is a function of the group communication service that is responsible for attaching a logical clock and forwarding it to other group communication services. For easy readability, only the invocation of `receive` of the group communication service of `S2` is shown in Figure 2. `send` and `receive` will be described in more detail in Section 4.
After the group communication service of `S2` has been called, it is forwarded to `S2` which performs the update in it's local database.

4 Group Communication in Jolie

The group communication services are embedded into the services that require it. The group communication service then provides `send` for the service to use, which ensures that the request to forwarded to other services using their respective group communication service. There are a number of parameters that are required when embedding the group communication service, as well as when invoking `send`. In addition to this, the service embedding the group communication service should also define a function called `update` which defines how to handle the incoming requests from the embedded group communication service. The following subsections will go into more detail about this.

4.1 Parameters

When embedding the group communication service, a number of parameters need to be set. This can be done by defining them in a JSON file.
The parameters needed for the group communication service are:

- `id`, which contains the id of the group communication service. This id is used for the logical clock.
- `logic`, which contains the location of the application embedding the given instance of the group communication service.
- `location`, which contains the location of the given instance of the group communication service.
- `others`, which is an array that contains the loctions of the other group communication services.

4.2 Interfaces

The group communication service uses two interfaces, the interface containing the functions that external services use, in the example in Figure 2 that would be `S1`, `S2`, or `S3` that is the external service. The other interface contains functions that internal services use, namely the other group communication services.

The external interface contains `send` operation. `send` broadcasts messages to all group communication services defined in the `others` parameter. `send` takes a request of type `sendMessageType`, which is defined in the following manner:

```
type sendMessageType: void {
  .data?: undefined
  .operation: string
}
```

The return type is `any` as it depends on what the functionality of the service embedding the group communication service.

The internal interface has the `receive` operation. `receive` adds the incoming message to the message queue and handles the delivery of the message to it's corresponding logic service. The `receive` operation takes a request of the type `receiveMessageType` which is defined in the following manner:

```
type receiveMessageType: void {
    .operation : string
    .data?: undefined
    .time : int
    .id: int
}
```

Again, the return type is `any` as it depends on what the functionality of the service embedding the group communication service.

4.3 Group Communication Service

The group communication service has two input ports and two output ports. As illustrated in Figure 1, the operations of the group communication service are exposed to both the service using the instance of the group communication service and other group communication services. Additionally, the service listens for requests from the service embedding it and other group communication services.
For the input port for the service using the instance of the group communication service it exposes the external interface containing the `send` operation, and for the input port for the other group communication services it exposes the internal interface containing the `receive` operation.

In `init` in the group communication service ensures that the location of the output ports of the service embedding the instance of the group communication service is updated to the right value. Moreover, in `init` the global variable for the local clock is initialized as an empty array, and the global variable for the local message queue is also initialized to be empty array as well.

In `main`, the functionality of two operations, `send` and `receive`, are defined. In the `send` operation the the first rule of the vector clock is performed, and after this the message being sent to the other group communication services is defined. The message includes the following information:

- `operation`, indicating the operation that was invoked in the service using the group communication service,
- `data`, containing the parameters and their values for the operation invocation,
- `time`, representing the timestamp for the request,
- `id`, containing the id of the group communication service

The message is sent to the other group communication services in the following manner:

```
spawn( i over #p.others ) in resultVar {
        GCOutput.location = p.others[i]
        receive@GCOutput( receiveRequest )( resultVar )
    }
```

It iterates over the locations given in the `others` parameter and dynamically rebinds the location of the output port `GCOutput`, which is the output port which the message is sent to. Additionally, the spawn ensures that all the responses are saved in the variable `restultVar`. It is up to the service embedding the group communication service to decide how to handle the multiple responses.

In the `receive` operation the incoming request is added to the message queue and second rule of the vector clock is performed. Updating its own clock value to the maximum of its current value based on the timestamp of the received message is done in the following manner:

```
for( i = 0, i < #p.others + 1, i++ ){
    if( request.time[i] > global.clock_value[i] ){
        global.clock_value[i] = request.time[i]
    }
}
```

After this, the delivery of messages to its service in which the instance of the group communication service is embedded is handled. This is done by calling the operation `update` for each message in the message queue where the time of the message at index *p.id* is less than or equal to the time of the local vector clock at index *p.id*. The code for this is:

```
for( i = 0, i < #global.queue, i++ ){
    if( global.queue[i].time[p.id] <= global.clock_value[p.id] ){
            update@LogicOutput( {operation = global.queue[i].operation, data <<
                global.queue[i].data } )( response )
    }
}
```

5 Evaluation of the Implementation

To evaluate the implementation of the group communication library, a basic server has been developed, as previously discussed in Section 3.
This basic server functions as a database that stores key-value pairs and provides basic functionality, namely `put`, `get`, and `delete`. The code for this server can be found in `/src`. As the functionality for the server is relatively simple and its purpose is solely to serve as a test environment to confirm the proper functioning of the group communication implementation the implementation of the server will not be discussed further in this report.

The form of testing has mainly been in the form of a client service and using the `dumpstate@Runtime` provided by Jolie to ensure the variables contain the expetced values. The client service used for testing will be available in `/src`. In the client service the tests have been of the form of invoking the operations `put`, `get`, and `delete` and ensuring that the response is the expected result.
An example could be that when calling the `put` operation on `S1`, it must be ensures that the changes can also be seen in `S2` and `S3` using the `get` operation on `S2` and `S3` on the newly added key-value pair. Similarly, the functionality of `delete` was tested in a similar manner, by invoking the operation on one of servers and seeing the changes on the others.

6 Jolie for Distributed Programming

The implementation of the group communication library in Jolie faced challenges due to the language being in development, meaning that at times limited documentation, which made it hard to fully utilize the language's capabilities. Additionally, sometimes there was some difficulty in understanding error messages.
When it comes to the programming language itself, the process of setting locations, either through hard coding or a parameter file, was also labor-intensive compared to other programming languages like Erlang. On the positive side, once the services were set up in Jolie, they could be relocated and replicated easily, by just adjusting the parameters in a file.

7 Conclusion

In this ISA a group communication library has been implemented in the service-oriented language Jolie. The group communication algorithm implemented is the multiple sources single group algorithm, with ensured causal ordering of messages and using broadcast as the method of message cast. When evaluating and testing the library, by using a simple implementation of servers, the library provides the expected results.

There are improvements to be desired for this library, one of them being the `update` function needed to be defined by the client. Perhaps a more elegant solution could have been achieved using some of functionality provided by Jolie such as the `invoke` from the `Reflection` interface, though it would not be possible to invoke `put`, `get`, and `delete` as is without creating an infinite loop.

References

[1] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed computing: principles, algorithms, and systems.* Cambridge University Press, Cambridge New York, 2012.