All-to-all Communication for Multiparty RPC

Vladyslav Maksyk
University of Stavanger, Norway
v.maksyk@stud.uis.no

ABSTRACT

Some protocols on distributed memory parallel computers can significantly benefit from an "all-to-all" communication pattern. This communication pattern should be implemented in a sequence of distinct phases, that assumes an order in which a process should operate. This paper introduces this communication pattern implemented in different design versions, including extensive benchmark evaluations.

KEYWORDS

Gorums, Quorum Call, Protocol Buffers, gRPC, Byzantine Fault-Tolerant, Paxos

1 INTRODUCTION

Cloud computing is taking over the world at an incredible pace [1]. In fact, according to research conducted by Cisco by 2021, only 6% of all compute instances will be processed by traditional data centers, the rest will move towards cloud computing.

To achieve fault-tolerance, these data centers deploy a variety of services, replicated using complex protocols. These protocols mainly rely on the quorum system to guarantee consistency. Quorum systems are an inevitable part of fault-tolerant systems. Quorum systems rely on the majority of the processes in the system to make progress. This means that the system can operate correctly as long as at most the minority of the processes fail.

Context and Motivation. Gorums is an RPC framework developed at UiS for building fault-tolerant distributed systems. Gorums allows us to perform quorum calls simultaneously on a group of hosts, collect and process their responses.

However, the bottleneck of the current version is that it only allows us to perform point-to-multipoint-to-point communication between the replicas. That is, a client invokes a request to all the servers in the system; each of the servers individually processes the request and sends a reply back to the client. This applies several limitations to the use of the protocol itself. It makes the protocol unsuitable for implementing (BFT) Byzantine Fault-Tolerant [2] and Paxos protocols.

Byzantine failures are the hardest failures to deal with. Byzantine fault-tolerance is a network feature built to reach consensus despite such failures. BFT implies no assumptions about the type of failure a node can have, meaning that a node can send any arbitrary data and still pose himself as an honest actor. Byzantine Fault-Tolerant protocols rely on an "all-to-all" communication pattern for the nodes to be able to communicate with each other and achieve the main goal, which is to reach an agreement between all the honest nodes.

Research Problem. This paper introduces a novel Gorums architecture that facilitates an all-to-all communication pattern. This feature will allow us to implement Byzantine fault-tolerant protocols, and we assume these protocols will greatly benefit prototyping and possibly even production quality protocol implementations. Furthermore, it could have a significant impact on the experimentation and evaluation of a variety of blockchain protocols.

Related Work. We refer the reader to the Gorums [3] paper to get a broad insight into the Gorums RPC framework. To the best of our knowledge, there is no other RPC framework that facilitates the "all-to-all" communication pattern.

Contribution Summary. The summary of our contributions is listed as follows:

- We implemented a novel Gorums architecture that facilitates the "all-to-all" communication pattern, enabling us to implement BFT protocols.
- We measured both the throughput and latency of the new architecture.

2 BACKGROUND

In this section, we are going to give the reader a superficial insight into Gorums [3] framework and its main features.

Gorums framework is built with the purpose to simplify the design and implementation of quorum systems by introducing a quorum call abstraction. Quorum call allows combining replies from a quorum into one single reply that contains the information about the whole system and ensures that replies from a quorum contain an up-to-date version of the data. This is handled by the quorum function. The quorum function remains to be a vital part of fault-tolerant systems as it decides whether the process received a sufficient amount of replies for a quorum call to succeed. To produce an RPC that can be used to invoke quorum calls, Gorums uses code generation. The code generation tool builds on existing toolchain consisting of Protocol Buffers and gRPC.

Protocol Buffers [4] are a simple, automated mechanism for serializing data. It allows you to structure your data and make use of automatically generated source code to write and read your data to and from a variety of data streams. It is primarily chosen for its encoding and decoding speed compared to other data-interchange formats such as JSON.

gRPC [5] is a remote procedure call framework used for data exchange between different processes. It uses protocol buffers as a default definition language and data structure. gRPC supports two main different communication styles:

Supervised by Hein Meling.

Project in Computer Science (DAT620), IDE, UiS
2019.

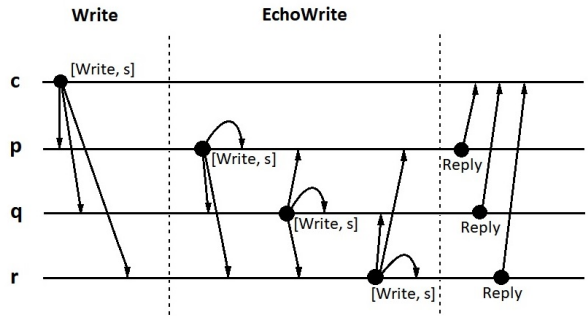

Figure 1: Sample execution of all-to-all communication for multiparty RPC

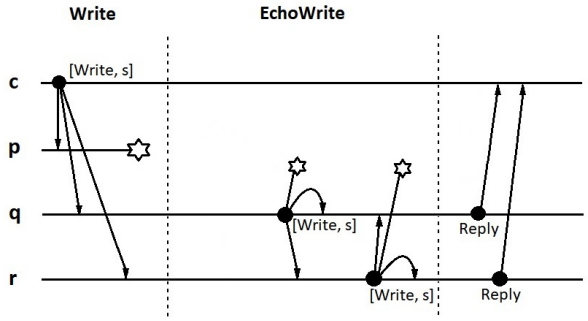

Figure 2: Sample execution of all-to-all communication for multiparty RPC with crash failure.

- A classic request-response REST-based API.
- Unidirectional or bidirectional streaming where the server streams large responses back to the requestor.

3 METHOD

To achieve the desired model, we used prototyping and experimenting with manually crafted code.

3.1 Design

Our environment consists of a dynamic set of processes where each process can act as a server or a client. Below in Figure 1, we show a sample execution of an "all-to-all" communication pattern for multiparty. Furthermore, we depict the communication process in two main phases, a *Write* phase, and an *EchoWrite* phase. For simplicity we use one writer process *c* (client) and three replicated servers *p,q,r* (server). The core ideas of the design are:

Initially, all correct processes store a timestamp(t) that is incremented every time a process invokes a *Write* or *EchoWrite* method. Timestamps are used by the quorum function to discard outdated messages. Client(c) invokes a *Write* method with (t) on all servers in the configuration. Consequently, each server initiates an *EchoWrite* method on all servers in the configuration, including itself, with its current timestamp. Each server processes the replies from other servers in the configuration using the quorum function. The *EchoWrite* operation is considered successful if the majority of the servers in the system are non-faulty and thus send a reply back. On completion of *EchoWrite*, each server sends a reply back to the client to indicate that the majority of the servers successfully executed the *Write* method.

In Figure 2, we show how the algorithm handles the crash of one replica. As an example of how the algorithm works, consider the following execution with a similar configuration of N = 4 processes, where (c) is the client and (p,q,r) are the servers. The sender process (c) invokes a *Write* on all servers in the configuration. Right after process (p) receives the *Write* message it crashes and thus terminates any further execution. As process (q) and (r) are correct, they receive the *Write* and proceed to the next phase where they invoke an *EchoWrite* on all initially registered servers. Since the majority of the servers are alive, processes (q) and (r) get enough replies to proceed to the final phase, where they send a reply back to the client. Only after the client receives the reply messages from (q) and (r) it informs itself that the quorum of the processes has replied to the *Write* and thus the execution of the algorithm completes.

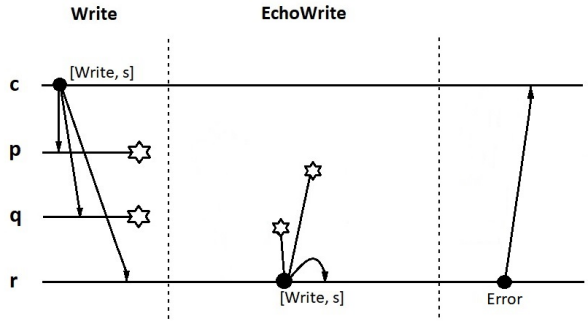

Figure 3: Sample execution of all-to-all communication for multiparty RPC with multiple crash failures.

Finally, in Figure 3 we show the algorithm execution when the majority of the replicas fail. Process (p) and (q) crush after receiving the Write message from (c). Only process (r) proceeds to the next phase. Process (r) invokes *EchoWrite* on all replicas in the configuration, but only gets a reply from itself. Since, process (r) only receives the minority of replies the quorum is not reached and thus process (r) returns a quorum call error to the client.

3.2 Implementation

In this section, we explain some key aspects of our implementation. In Listing 1 we define the mechanism for data transmission the Protocol Buffers with gRPC specifications as a form of remote procedure calls. We create two services called *Write* and *EchoWrite* that have *rpc* methods. In addition we specify two *gorums* options:

- ***gorums.qc*** We call all servers in the quorum, the quorum function gets called for each reply, and eventually signals that the quorum call is done and returns to the client.
- ***gorums.qf-with-req*** Allows the quorum function to take the request object as its first parameter. This is used by the quorum function.

Finally, both methods require an input and output message to be specified.

```
service Storage
  rpc Write(Value)returns(WriteResponse)
    option(gorums.qc)=true
    option(gorums.qf_with_req)=true

  rpc EchoWrite(Value)returns(WriteResponse)
    option(gorums.qc)=true
    option(gorums.qf_with_req)=true
```

Listing 1: Defining protocol buffers data structure

In Listing 2, we define protobuf messages as structured objects. Where *Value* is the input to the *rpc* methods and *WriteResponse* is the output.

```
message Value
    string key = 1
    string value = 2
    int64 timestamp = 3

message WriteResponse
    int64 timestamp = 1
    int64 port = 2
```

Listing 2: Defining protocol buffers data structure

After compiling the *proto* file to generate data access classes, we define the functions *Write* and *EchoWrite* on the server side that will be called by the service *Storage* from the *proto* file. In addition, we define *storage struct* on the server side that will allow us to provide resources to the server, making them available for the rpc calls.

At the start, each server attempts to connect to all other servers in the configuration using a *gRPC* Dial option *WithBlock*. This option blocks the server's dial to other servers until they become available. When that is done, the servers move to a *listen* state when they are ready to accept and process client requests.

Algorithm 1: Write calls an EchoWrite

```
storage struct ← state

Function Write(value):
    ack, err ← EchoWrite(state)
    resp ← WriteResponse(value.Timestamp)
    return resp, nil

Function EchoWrite(value):
    resp ← WriteResponse(value.Timestamp)
    return resp, nil
```

In Algorithm 1 we show how the server invokes *EchoWrite*. When the server receives a client *Write* request, it immediately sends an *EchoWrite* request to other servers. We use a user-defined quorum function to process the replies and determine if a quorum of replies has been collected. On completion, the server sends a reply back to the requestor. Finally, the *client* uses the same quorum function

Algorithm 2: Quorum Function

```
Function WriteQF(req Value, replies []WriteReply):
    if len(replies) < quorum then
        return nil, false
    correctReplies ← ∅
    for r ← range replies do
        if r.timestamp == req.timestamp then
            correctReplies ← correctReplies + 1
            reply ← r
    if correctReplies < quorum then
        return nil, false
    return reply, true
```

to process the replies from the servers and determine whether the quorum of the servers replied.

In Algorithm 2, we show the Quorum Function. It is invoked every time a process gets a reply, and it has two main objectives. Firstly, it collects all replies until the quorum of replies is reached. Secondly, it combines all replies into one and returns the last received reply. The Quorum function also compares the timestamp of the reply to ensure it is up to date. The function keeps count of the correct up to date replies, and when they reach the quorum, it returns a *true* value indicating that the quorum has been reached.

4 EXPERIMENTAL EVALUATION

In this section, we present and describe the experimental setup and our results.

4.1 Experimental setup

We run our experiments on a distributed system cluster. A distributed systems cluster is a group of machines that are virtually or geographically separated, and that work together to provide the same service or application to clients [6]. The hardware is managed by the Linux operating system and consists of multiple processors(80) with a 2.5 GHz frequency that can operate independently on shared memory. We generate the load by sending multiple Write requests in an open-loop from the client to imitate an unbounded amount of clients and then measure the performance of different design versions. The instructions on how to reproduce the experiments and the source code will be freely available on our Github [7]. Please follow the `README.md` file for detailed step-by-step instructions.

4.2 Results

In this section, we provide a detailed benchmark evaluation of the built architecture. We focused in particular on throughput and latency metrics. In Figure 4 and Figure 5 we show the throughput of the three different design variants:

- ***Write.*** A usual scenario of message exchange between client and multiple servers that uses "point-to-multipoint-to-point" communication.

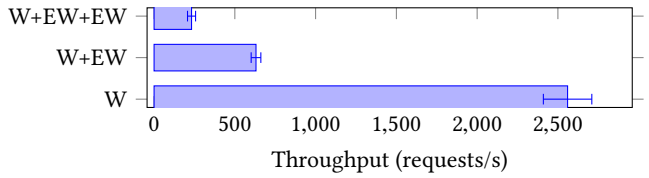

Figure 4: A barplot showing throughput of a configuration with 4 servers. Average of 10 runs with 95% confidence interval.

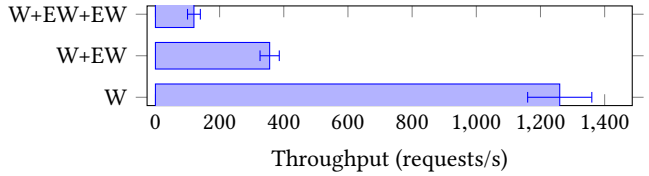

Figure 5: A barplot showing throughput of a configuration with 7 servers. Average of 10 runs with 95% confidence interval

- ***Write + EchoWrite.*** Scenario that assumes "all-to-all" communication with an *EchoWrite* message exchange among the servers.
- ***Write + EchoWrite + EchoWrite.*** Scenario that assumes "all-to-all" communication where each *EchoWrite* message is followed by another *EchoWrite* message between the servers.

Based on the result, we can state that every additional *EchoWrite* significantly reduces the throughput due to an extra set of messages being exchanged.

In general, throughput measures the volume of traffic that a server handles. Figure 7 shows the correlation between latency and throughput. The growth of throughput makes latency increase along. This happens because, under higher loads, the server becomes slower than in normal load. With normal loads, the latency tends to be more consistent despite throughput fluctuations Figure 6. When the throughput reaches a certain bound, the latency starts to increase linearly and sometimes exponentially. And when throughput reaches the limit, the latency can increase drastically. The limit is determined by the hardware and the network.

5 FURTHER WORK

Further work would be to build an automatic code generator. This is needed to generate the code that was manually written in order to reduce the effort for the developer to build such protocol. Furthermore, we consider it possible to create a new Gorums option that will allow us to define a *Write* RPC without the need for defining an *EchoWrite* additionally. We think that this feature can be built-in, and should not necessarily be visible to the developer unless he wants to do something different on the server. It should be possible to have an arbitrary number of *EchoWrites* ($Write_1$, $EchoWrite_2$,..., $EchoWrite_n$) with the ability to specify the exact number. Eventually, we would like to implement a protocol like Paxos or PBFT that requires all-to-all communication.

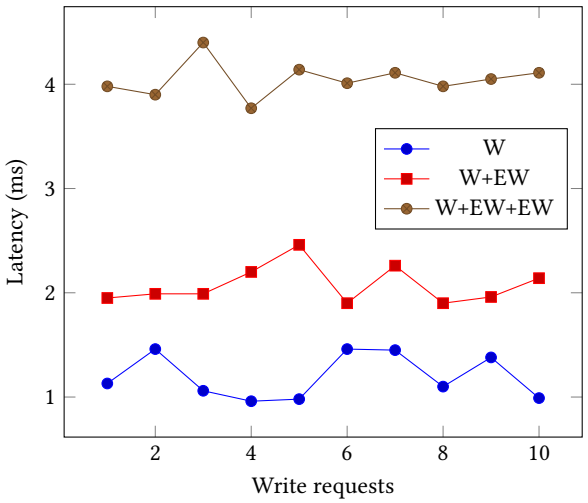

Figure 6: A graph showing latency in milliseconds for a configuration of 4 servers and 1 client throughout 10 different runs.

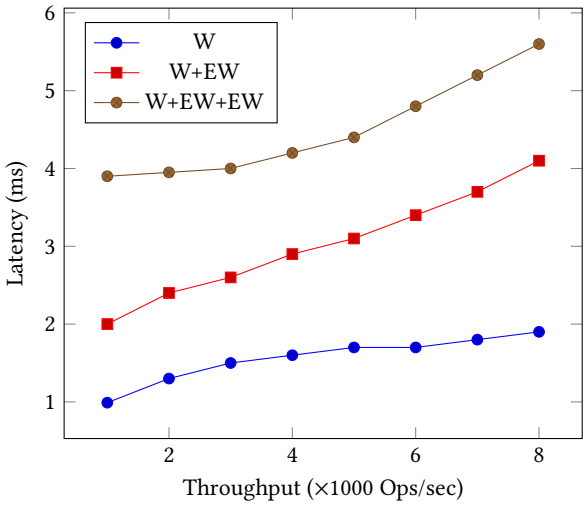

Figure 7: A graph showing latency and throughput. The axes show latency in seconds and throughput in thousand operations per second.

6 CONCLUSION

After experimenting and prototyping with manually crafted code, and after extensive evaluation, we built a novel Gorums architecture that facilitates "all-to-all" communication pattern. This communication pattern opens up the possibility to build PBFT and Paxos protocols.

REFERENCES

[1] The era of the cloud's total dominance is drawing to a close, 2018. URL https://www.economist.com/business/2018/01/18/the-era-of-the-clouds-total-dominance-is-drawing-to-a-close.

[2] Brian Curran. What is practical byzantine fault tolerance?, 2018. URL https://blockonomi.com/practical-byzantine-fault-tolerance/.
[3] Hein Meling, Tormod Erevik Lea, and Leander Jehl. Towards new abstractions for implementing quorum-based systems. 6(3):1–6, 2017. doi: 10.1109/ICDCS.2017.166. URL https://ieeexplore.ieee.org/document/7980198.
[4] Protocol Buffers, 2019. URL https://developers.google.com/protocol-buffers/docs/overview.
[5] Andrew Connel. grpc and protocol buffers: an alternative to rest apis and json, 2016. URL https://www.andrewconnell.com/blog/grpc-and-protocol-buffers-an-alternative-to-rest-apis-and-json/.
[6] Joseph M. Lamb. Clustering and load balancing, 2002. URL http://www.informit.com/articles/article.aspx?p=25748&seqNum=4.
[7] Vladyslav Maksyk. All-to-all communication for multiparty rpc, 2019. URL https://github.com/vladmaksyk/byzq-master/tree/master/byzq-master.