Individual Diploma Thesis

IMPLEMENTATION AND VISUAL REPRESENTATION OF BYZANTINE-TOLERANT DISTRIBUTED ALGORITHMS FOR THE ATOMIC APPENDS PROBLEM

Marios Nicolaou

UNIVERSITY OF CYPRUS

Department of Computer Science

May 2023

University of Cyprus

Department of Computer Science

Implementation and Visual Representation of Byzantine-Tolerant Distributed Algorithms for the Atomic Appends Problem

Marios Nicolaou

Supervisor

Chryssis Georgiou

The Individual Diploma Thesis was submitted for partial fulfillment of the requirements for obtaining the degree of Computer Science of the Department of Computer Science of the University of Cyprus.

May 2023

Acknowledgements

Firstly and most importantly, I would like to express my honest gratitude towards my supervisor, Professor Chryssis Georgiou, who gave me the opportunity to work on this project. During the past year, he was always willing to help me with any question that I had, either with the implementation or with the theoretical aspect of the project. We also held weekly meetings during the semesters, which helped a lot with the progress of our Thesis.
Lastly, I would like to thank the Computer Science Department of the University of Cyprus, which provided us with such courses, that allowed me to achieve the best result I could for this project.

Abstract

During recent years, there has been a massive interest in *Distributed Ledger Technology* (Blockchains). Distributed Ledgers are decentralized synchronized databases that operate across multiple nodes or entities. They utilize technologies like blockchain to provide transparency and security without relying on a central authority. Distributed Ledgers enable direct client interactions and ensure data integrity through consensus mechanisms. To this respect several algorithms have been devised either for designing distributed ledgers or for applications across ledgers. In this study we implement two distributed algorithms designed to solve the *Atomic Appends problem*.

The Atomic Appends problem emerges when several clients have records to append, the record of each client has to be appended to a different Distributed Ledger, and it must be guaranteed that either all records are appended or none.

The general solution of both algorithms is the use of a distributed reliable middleman service, which is trustworthy and tolerates *Byzantine faults*. Byzantine faults refer to the arbitrary behaviour displayed by components in a Distributed Computing System, where these components might send misleading information to other components, which makes it very difficult to achieve consensus.

This study extends the study of Andreas Chrysanthou, who implemented a Graphical User Interface for a *Byzantine Tolerant Distributed Ledger Object*.

Contents

Chapter 1

Introduction

1.1 Motivation

In recent years the race towards trusted and secure blockchain systems [12], and *Distributed Ledger Technologies* (DLTs) in general, has gathered a lot of interest mostly due to the advances in cryptocurrencies such as Bitcoin [17]. This interest brought new innovations and variations of DLTs.

One of these innovations is *Distributed Ledger Object* (DLO) [9], which encapsulates the most essential elements of a blockchain. A DLO supports two operations, APPEND and GET. The first adds a new record at the end of the ledger (sequence of records), and the second one returns the sequence of records.

In [10], the authors initiate a study of systems that consist of multiple DLOs that interact with each other. In this study they define the *Atomic Appends problem* [3, 10] where a number of clients have records to append to different DLOs, and it must be guaranteed that either all records are appended or none. This guarantee must work with consensus in mind, in order to tolerate Byzantine faults [3]. The authors come to the conclusion that in order to solve the Atomic Appends problem, the use of a middleman is necessary.

The rise of digital goods and assets, like Non-Fungible Tokens (NFTs) [11] and cryptocurrencies, has made the development of a system that allows clients to exchange assets securely and fairly, very critical. In this study we build an asynchronous and *Byzantine Fault-Tolerant* system that solves the Atomic Appends problem.

1.2 Objective and Contribution

In [3], the authors propose two algorithms which use a distributed byzantine-tolerant middleman service in order to solve the Atomic Appends problem. The names of these algorithms are *Helper Processes* and *Smart Byzantine Distributed Ledger Object* (smart BDLO). The two algorithms are very similar to each other, with the main difference being that the middleman is an independent entity in the Helper Processes algorithm, whereas in Smart BDLO, the middlemen is integrated within the BDLO.
The purpose of this study is to implement these algorithms in Java Programming Language [1]. To implement the algorithms we extend Antdeas Chrysanthou's study [2], where he implements a graphical user interface for an asynchronous DLO, which is resilient to Byzantine faults.

1.3 Methodology

At first we studied the work of Andreas Chrysanthou [2] who developed a prototype of an *unbounded Byzantine Distributed Ledger Object* (unbounded BDLO) [3]. With the term unbounded we mean that there is no limit on how many clients can be Byzantine. At the same time we studied some articles on Asynchronous Byzantine Distributed Ledgers [3, 9, 10] to get more familiar with the subject.
Then we were facing the dilemma to either recreate Andreas Chrysanthou's application on Go Programming Language and expand it, or build on the existing code which was already written in Java from Andreas. For practical reasons, we chose to build on Andrea's work since we are more familiar with Java, and it would require much more effort and time to start over with Go.
Later on, we made sure that the application was working as it should, even when the servers (that consisted of the BDLO) were not on the same network (this was tested using Virtual Machines) – and it did.

Then we realised that in order to implement the two algorithms of Atomic Appends, we first had to develop a version of a BDLO called bounded BDLO, which is a different variation of what was implemented in [2]. As the name suggests, a bounded BDLO limits the number of clients that can be Byzantine. We started working on the implementation in November, and it was completed at the beginning of December.

The next step was to implement the first algorithm, which was the Helper Processes algorithm. The main reason we chose to develop this algorithm first, was because it was practically harder to implement and it would make easier the implementation of the Smart BDLO algorithm. The implementation and testing started in the middle of January, and it was completed in the middle of March. It took more time than anticipated, with reason being that we faced some incompatibilities with the existing code (more information is given in Sections 4.5 and 6.2).

Subsequently, in late March, we begun developing the Smart BDLO algorithm, which was completed much faster, due to the fact that part of the implementation was already done in the Helper Processes code.

After testing the Smart BDLO prototype, we decided to test the Helper Processes algorithm on Emulab [6], to verify that it was working on servers that were not on the same network.

The writing of the Individual Thesis started from the beginning of Fall semester. However that was postponed when the development of the algorithms started, and it was resumed in the beginning of April once the Emulab testing was completed. In Figure 1.1 we present a Gantt diagram which we used to plan and schedule our tasks.

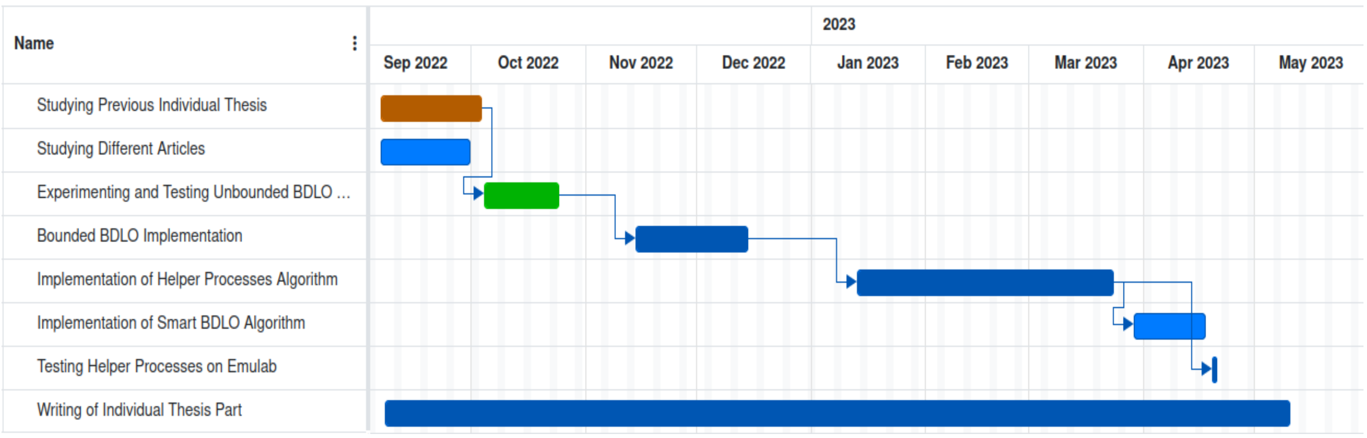


Figure 1.1 : Gantt Diagram of our Scheduling

1.4 Document Organization

In **Chapter 2,** we provide definitions and background information necessary for the material presented in later chapters. We also overview the BFT-Smart library that was used for the implementation of the algorithms, and an overview of Andreas Chrysanthou's study.
In **Chapter 3**, we present the two Atomic Appends Algorithms, the Helper Processes and Smart BDLO and in **Chapter 4**, we provide the implementation of the two algorithms.
In **Chapter 5**, we provide a Visual Representation of both implementations, and an experimental evaluation of the Helper Processes algorithm on Emulab [6].
In **Chapter 6**, we conclude with a summary of the study, a discussion on the challenges addressed and possible future extensions of the work.

Chapter 2

Background

2.1 Byzantine Fault Tolerant System

Byzantine Fault Tolerance [3] emerges from the *Byzantine Generals Problem* [13]. In this hypothetical problem a Byzantine Army is divided into *n* separate divisions and each one is controlled by a General. All divisions have to make a common decision whether or not they will attack the enemy. This is also called *Byzantine Consensus* [14]**.** The problem appears when we assume that some generals can be traitors/malicious. In this scenario, the malicious general will tell to some generals to attack and to others to retreat. This results as a defeat for the army. Keep in mind that when a General receives ambiguous orders, they retreat. They will attack only when all orders are ATTACK. They only win the battle, if every general decides to attack.

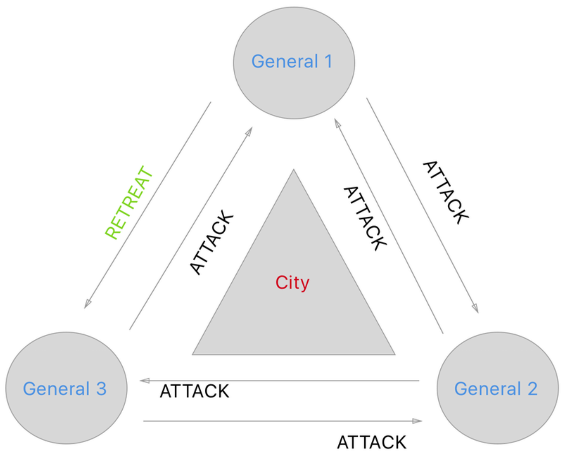


Figure 2.1: Byzantine Generals Problem with n = 3

In Figure 2.1, an example of this problem with 3 Generals is shown. For this situation, General 1 is the malicious one. General 1 sends to General 2 Attack order, but to General 3 Retreat order. General 2 and 3 send to everyone Attack order. So General 3 has one Retreat and one Attack order. So from above, we understand that General 3 will not attack. General received 2 Attack orders. So General 2 will attack. Therefore the battle will be lost.

It has been proven that this problem has no solution, and Consensus can only be achieved when there are at least n generals and f of them are malicious, where $n \geq 3f + 1$.

Byzantine Fault Tolerance (BFT) is the attribute of Distributed Systems that enables them to reach Consensus even in the presence of Byzantine Faults. A Byzantine Fault is considered a process (or a node), which deviates from its assumed behaviour based on the algorithms it's running. The goal of Byzantine Fault-Tolerant systems, is to function how they normally would, even when 1/3 of the nodes act maliciously or arbitrarily. Byzantine Fault Tolerant Systems and algorithms usually need make this assumption in order to operate as they are meant to.

2.2 Distributed Ledger Objects

A *Distributed Ledger Object* (DLO) [9], as the name suggests, is a ledger that is distributed to different servers. Each server has an instance of the Ledger, and the records on each one of them must be the same and in a totally ordered sequence. A DLO L has two main functionalities.

- ***L.append(r) :*** The Record $r = <\tau,p,v>$ will be appended onto the Ledger, where τ is a unique identifier of the record (maybe the hash code of p and v), p is the identification of the process that appended the record and v is the value of the record.
- ***L.get() :*** Will return the sequence S of records that are currently in the Ledger will be returned.

2.3 Byzantine Tolerant Distributed Ledger Object

A *Byzantine-tolerant DLO* (BDLO) [3], is a DLO which is able to function normally even with the presence of Byzantine Faults. A server is considered Byzantine when it acts arbitrarily, maliciously or doesn't work at all. At any given time, for the BDLO to work correctly, the servers n, must be $n \geq 3f+1$ where f are the Byzantine servers. Also a BDLO must satisfy 3 properties [3]:

- *Byzantine Completeness* (BC): All the GET and APPEND requests from the clients, eventually are completed.
- *Byzantine Strong Prefix* (BSP): If two clients C and C' send a get request to the DLO then if the returned sequence of records are S and S' respectively, then S is a prefix of S' or S' is a prefix of S.
- *Byzantine Linearisability* (*BL*): Let G be the set of all complete get operations issued by correct clients. Let A be the set of complete append operations L.append(r) such that $r \in S$ and S is the sequence returned by some operation L.get() $\in G$. Then linearisability holds with respect to the set of operations $G \cup A$.

2.4 The Atomic Appends Problem

The *Atomic Appends* problem [10] can be considered as a specific version of the fair exchange problem. In the Atomic Appends problem, several clients have records to append, and each record must be appended in a different DLO with the use of a middleman. It must be guaranteed that either all records are appended to their respective DLOs, or none. In our study we implement a specific version of the Atomic Appends problem, where only 2 clients come into an agreement.

Example:

Let us consider clients p and q. Client p has a car that client q wants to buy using cryptocurrency. For the car to be in the ownership of q, a record r must be appended on the DLO $L_{\mathbf{car}}$ from client p. For the client p to receive an amount of cryptocurrency from q, then q must append a record r' on the DLO $L_{\mathbf{crypto}}$.

Cases

- In the perfect scenario both clients abide by their agreement, and they append the correct records to the middleman. Then the middleman verifies that both clients have appended their records and continues by appending each one of them to their appropriate DLO.
- In this scenario, client p appends its record to the middleman. However, client q pulls out of the agreement and doesn't append its record. The middleman verifies that not all clients have appended their records, hence it doesn't append the record of client p in its respective DLO.

In Chapter 3 we present two algorithms that solve the Atomic Appends problem as they are proposed in [3].

2.5 Byzantine Atomic Broadcast and BFT-Smart

2.5.1 Byzantine Atomic Broadcast

Atomic Broadcast [16] is a messaging protocol that is used in distributed systems. Its goal is to ensure that a message is delivered to all the nodes of a network. In an atomic broadcast system, all messages are delivered to all participants in the same order, maintaining consistency across all nodes. In our case, the atomic broadcast is implemented and extended from the BFT-Smart library [5, 15] which makes it tolerable to Byzantine failures. Hence it is called Byzantine Atomic Broadcast (BAB).

In [3], the authors use a BAB service for the server communication in order to solve the Atomic Appends problem. This service satisfies the following properties:

- Validity : if a correct server BAB-broadcasts a message, then it will eventually BAB-deliver it.
- *Agreement* : if a correct server BAB-delivers a message, then all correct servers will eventually BAB-deliver that message.
- *Integrity* : a message is BAB-delivered by each correct server at most once, and only if it was previously BAB-broadcast.
- *Total Order* : the messages BAB-delivered by correct servers are totally ordered; i.e., if any correct server BAB-delivers message m before message m', then every correct server must do it in that order.

2.5.2 BFT-Smart

BFT-Smart is an open source library written in Java and its purpose is to replicate Byzantine fault-tolerant machines. The project was implemented by researchers at the University of Lisbon and we can find it in [5]. The library makes use of multithreading and it's relatively easy for someone to extend it using the API provided [15].
Developers implement specialized behaviours using a set of alternative calls, callbacks or plug-ins both at client and server-side. In the thesis of Andreas Chrysanthou [2], the library was extended in order to implement a BDLO, which consists of servers and clients.

BFT-Smart Architecture

In Figure 2.2 we can see the architecture of BFT-Smart .

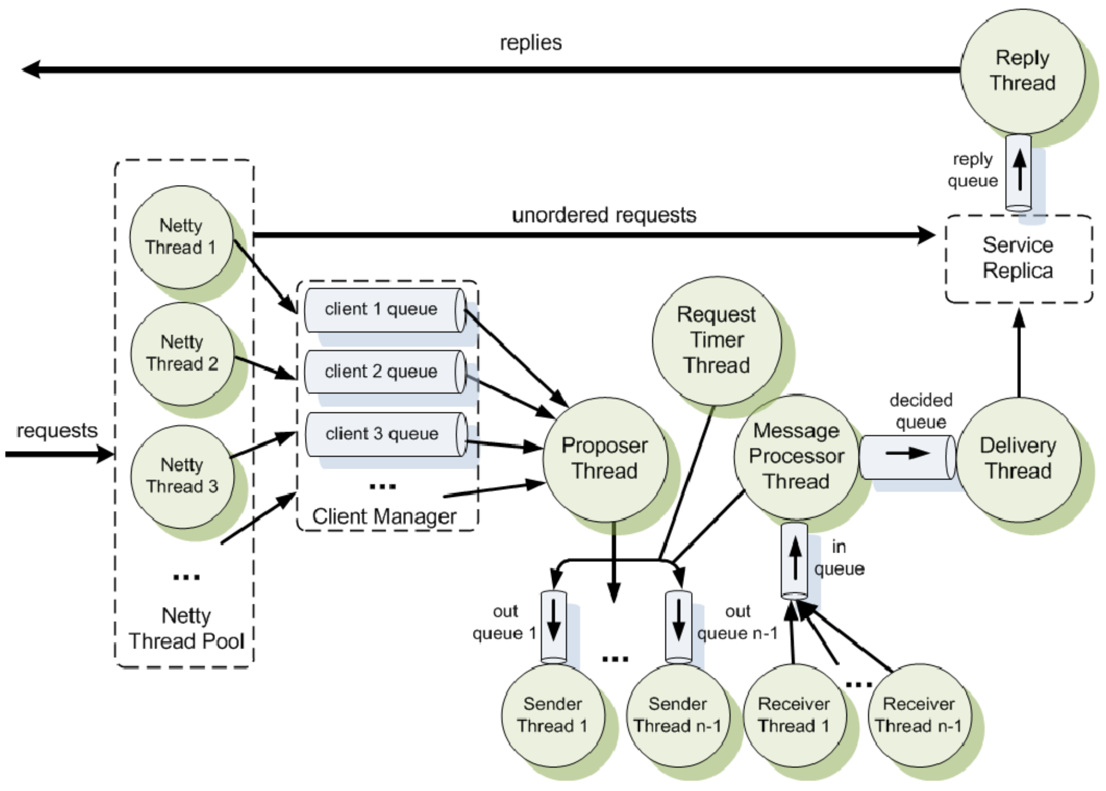


Figure 2.2 : BFT-Smart Architecture [5]

All communication between threads in this architecture occurs via queues with a limited capacity. The diagram illustrates which queues are responsible for data input and output for each thread.

A thread pool offered by the Netty communication framework handles the requests made by clients. Upon receiving a message from a client, the system first determines if it is an ordered or unordered request. If it is an unordered request, typically used for read-only operations, the request is directly sent to the service implementation. On the other hand, if it is an ordered request, it is forwarded to the client manager, which verifies the request's integrity and adds it to the appropriate queue for that particular client.

The proposer thread is in charge of collecting a group of requests and sending the Propose message to initiate the consensus protocol. In BFT-Smart, the batch is populated with requests until it reaches a maximum size specified in a configuration file or there are no more requests to add. This particular thread is only active in the leader replica.

To send a message *m* from one replica to another, it is first placed in an outgoing queue. A sender thread retrieves the message, serializes it, generates a MAC (message authentication code), and sends it over TCP sockets. At the recipient replica, a corresponding receiver thread reads the message, authenticates it, deserializes it, and stores it in the incoming queue. All received messages from other replicas are stored in this queue to be processed in sequence.

The task of handling messages from the BFT SMR protocol falls on the message processor thread. This thread retrieves messages from the incoming queue and processes them only if they belong to the ongoing consensus. If messages belong to a consensus ahead of the current one, they will be processed later, when that consensus is initiated. If messages do not fit into either of these categories, they are disregarded.

Once a consensus is completed on a replica, the confirmed batch is placed on the "decided" queue. The delivery thread is responsible for obtaining batches from this queue, deserializing all requests in the batch, removing them from their corresponding client queues, and marking the present consensus as completed. Following that, the delivery thread calls the service replica to execute the requests and produce the corresponding responses. Once the reply is generated, the service replica puts it into the reply queue. The reply thread retrieves replies from this queue and sends them to the corresponding clients.

BFT-Smart Configuration

There are two files that need to be configured in order for the library to be working as expected. These two files are hosts.config and system.config.

The hosts file, is used to configure the IP addresses and ports of each server that belongs to the system. In Figure 2.3, we can see the structure of the file.

```
#server id, address and port (the ids from 0 to n-1 are the service replicas)
0 155.98.37.45 11000 11001
1 155.98.37.64 11010 11011
2 155.98.37.50 11020 11021
3 155.98.37.49 11030 11031
4 155.98.37.47 11040 11041
```

Figure 2.3: Hosts.config Structure

The first column corresponds to the id of the server, the second to the IP address, the last two to the ports which are used to communicate with the rest of the replicas.

The system file is used to configure various functionalities and options of the library. Table 2.4 contains the most important settings that are needed in our scenario.

	Settings	Description
1	system.communication.bindaddress	The IP address that the replica binds to
2	system.servers.num	Number of servers in the group
3	system.servers.f	Maximum number of faulty replicas
4	system.initial.view	Replicas ID for the initial view, separated by a comma (i.e. 1,2,3,4 for servers.num = 4)
5	system.bft	For our case this needs to be set to true
6	system.communication.defaultkeys	Force all processes to use the same public/private keys pair and secret key. For testing purposes this must be set to true, otherwise must be false for security purposes

Table 2.4 : Most important parameters in system configuration file

2.6 Previous Thesis

The previous thesis [2], had as purpose to create a working prototype which implemented Byzantine Distributed Ledger Object. The author extended the BFT-Smart library by writing the full stack code for server and client side. The experimental evaluation of the thesis, demonstrated that the BDLO properties, were satisfied only when $n \geq 3f + 1$.

In Figure 2.5 we can see a visual representation of how a client communicates with the servers, and how the servers communicate with each other within a BDLO. At first a client sends a request to the first available server. On the second step the request is broadcasted to the rest servers using the BAB. Then all servers (except N3 which is Byzantine) , communicate with each other in order to verify that at least $f + 1$ of them have received the same request. In this case, it’s true so all the servers commit the request that they received earlier. At last, if the request was executed, the client receives an acknowledgement that their request was completed.

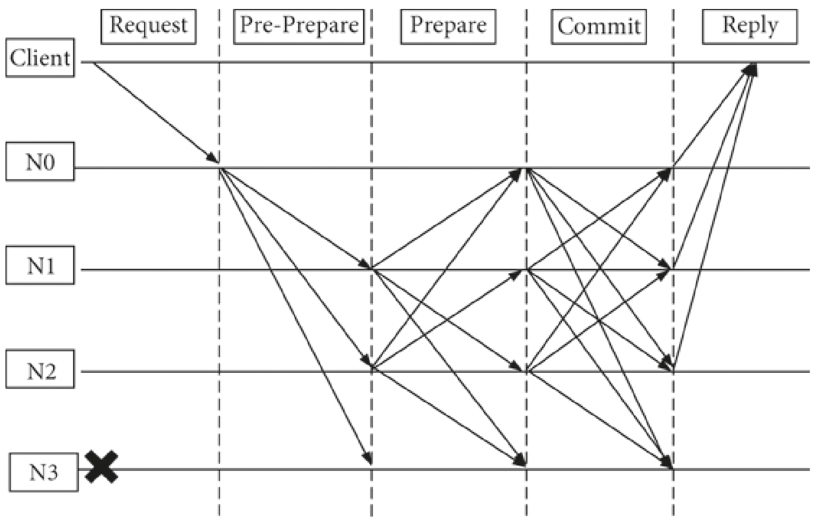
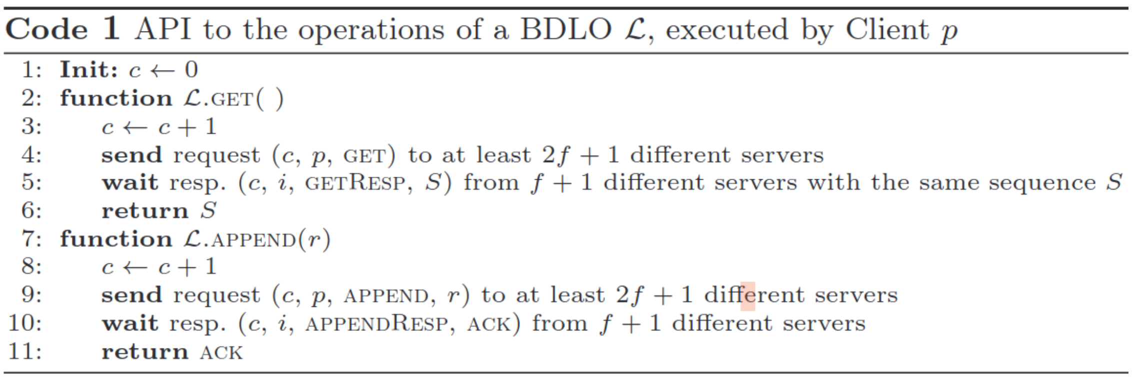


Figure 2.5 : A visual representation of Clients Server Communication in a BDLO

2.6.1 Unbounded BDLO

The version of the BDLO that was implemented, is called unbounded BDLO. With the term unbounded, we mean that there is no limit on how many clients can be Byzantine. We proceed to present the algorithms for the clients and the servers of an unbounded BDLO.

The clients p is connected to the unbounded BDLO L, and it wants to append the record r. The BDLO consists of *n* servers of which *f* of them are considered malicious, where $n \geq 3f + 1$. The client, runs the code shown in Figure 2.6.

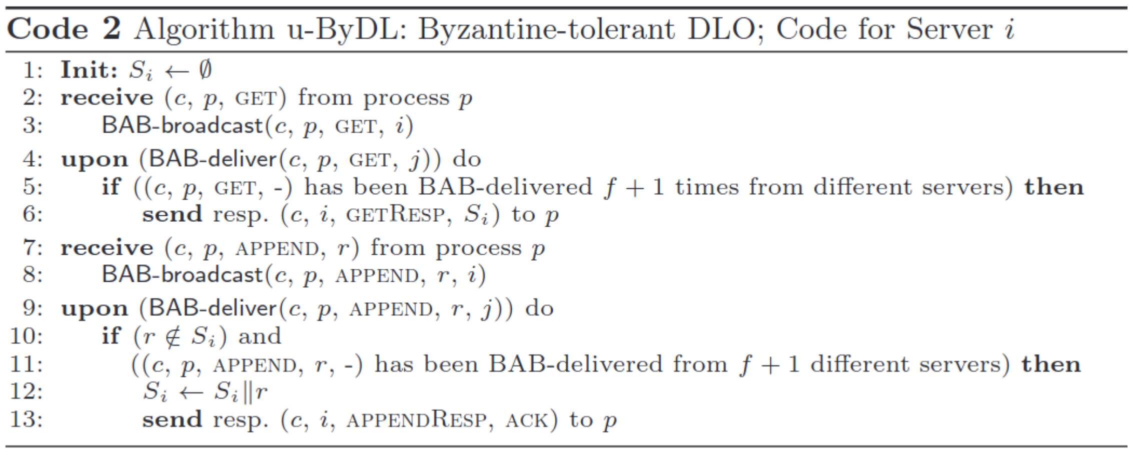


```
Code 1 API to the operations of a BDLO $\mathcal{L}$, executed by Client $p$
 1: Init: $c \leftarrow 0$
 2: function $\mathcal{L}$.GET( )
 3:     $c \leftarrow c + 1$
 4:     send request ($c$, $p$, GET) to at least $2f + 1$ different servers
 5:     wait resp. ($c$, $i$, GETRESP, $S$) from $f + 1$ different servers with the same sequence $S$
 6:     return $S$
 7: function $\mathcal{L}$.APPEND($r$)
 8:     $c \leftarrow c + 1$
 9:     send request ($c$, $p$, APPEND, $r$) to at least $2f + 1$ different servers
10:     wait resp. ($c$, $i$, APPENDRESP, ACK) from $f + 1$ different servers
11:     return ACK
```

Figure 2.6: Client operations pseudocode [3]

When a client wants to perform an APPEND operation, it first increases a counter and then sends request to a minimum of 2*f* + 1 (lines 8 and 9) servers to ensure that at least *f* + 1 correct servers receive the request. For a GET operation, it also increases the counter, and sends the request to a minimum of 2*f* + 1 servers (lines 3 and 4). T*he* client receives the sequence S, only if it receives *f* + 1 same replies of the sequence S (line 5). For an append operation, the client considers it complete when it receives *f*+1 replies from different servers (line 10). In both cases, the response from at least one correct server is guaranteed. The counter is used to give a unique ID for every request.

The servers of the unbounded version run the code shown in Figure 2.7



```
Code 2 Algorithm u-ByDL: Byzantine-tolerant DLO; Code for Server $i$
 1: Init: $S_i \leftarrow \emptyset$
 2: receive ($c$, $p$, GET) from process $p$
 3:     BAB-broadcast($c$, $p$, GET, $i$)
 4: upon (BAB-deliver($c$, $p$, GET, $j$)) do
 5:     if (($c$, $p$, GET, -) has been BAB-delivered $f + 1$ times from different servers) then
 6:         send resp. ($c$, $i$, GETRESP, $S_i$) to $p$
 7: receive ($c$, $p$, APPEND, $r$) from process $p$
 8:     BAB-broadcast($c$, $p$, APPEND, $r$, $i$)
 9: upon (BAB-deliver($c$, $p$, APPEND, $r$, $j$)) do
10:     if ($r \notin S_i$) and
11:        (($c$, $p$, APPEND, $r$, -) has been BAB-delivered from $f + 1$ different servers) then
12:         $S_i \leftarrow S_i \| r$
13:         send resp. ($c$, $i$, APPENDRESP, ACK) to $p$
```

Figure 2.7 : The algorithm the servers run on unbounded-BDLO [3]

The algorithm relies on the BAB service, which is implemented by the BFT-Smart library [15], to establish a complete order of messages exchanged among the servers. Whenever a client sends an operation to the servers, it is BAB-broadcasted using the

BAB service and is BAB-delivered eventually. A server processes an operation only when it has been BAB-delivered $f + 1$ times by different servers, which ensures that at least one correct server has sent the operation. The BAB service ensures that all the correct servers receive the same sequence of messages, which are BAB-delivered, therefore maintaining their state consistent and enabling them to process the operations at the same point.

2.6.2 Visual Representation of the Prototype

In this section we present a visual representation of the implementation. As mentioned before, a graphical user interface was developed both for the servers and clients. In Figure 2.8 we can see the GUI of a client.

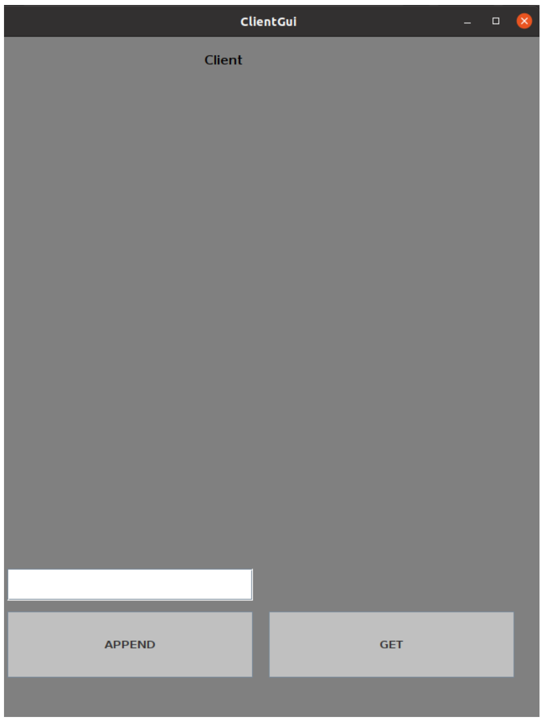
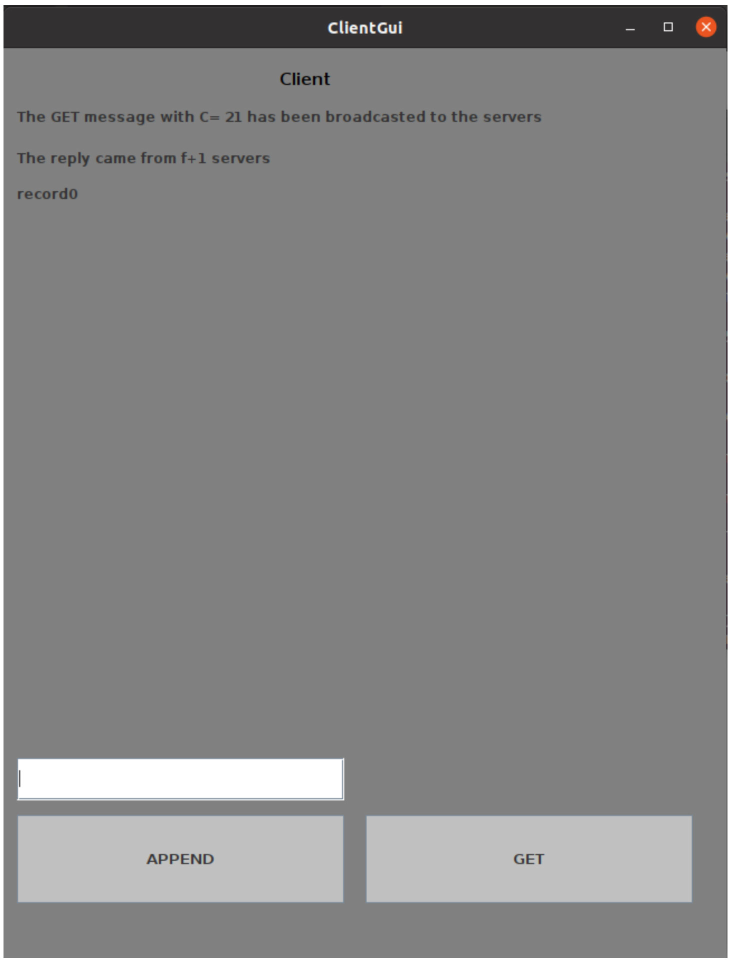


Figure 2.8 : Graphical User Interface of a Client

The client's graphical user interface has two buttons, which represent the two main functionalities of appending a record, and reading the sequence of records from the BDLO. It also includes a text box, in which the clients insert the value of the record that they want to append.

Say that the client wants to append a record on the ledger with the value record0. They must write in the text box the string record0, and then click append. The request successful only if at least $f + 1$ servers receive the request and append the record. The client doesn't know what happens once it executes the APPEND request. For the client the BDLO is a black box. Figure 2.9 shows the GUI of the client when it executes a GET request, after the append request has been completed.



The GET message with C= 21 has been broadcasted to the servers

The reply came from f+1 servers

record0

Figure 2.9 : Client GUI, after GET request, once record0 is appended

In Figure 2.10, we see the GUI of a server when the append request for the record is completed. All non-Byzantine servers have appended the record with value record0.

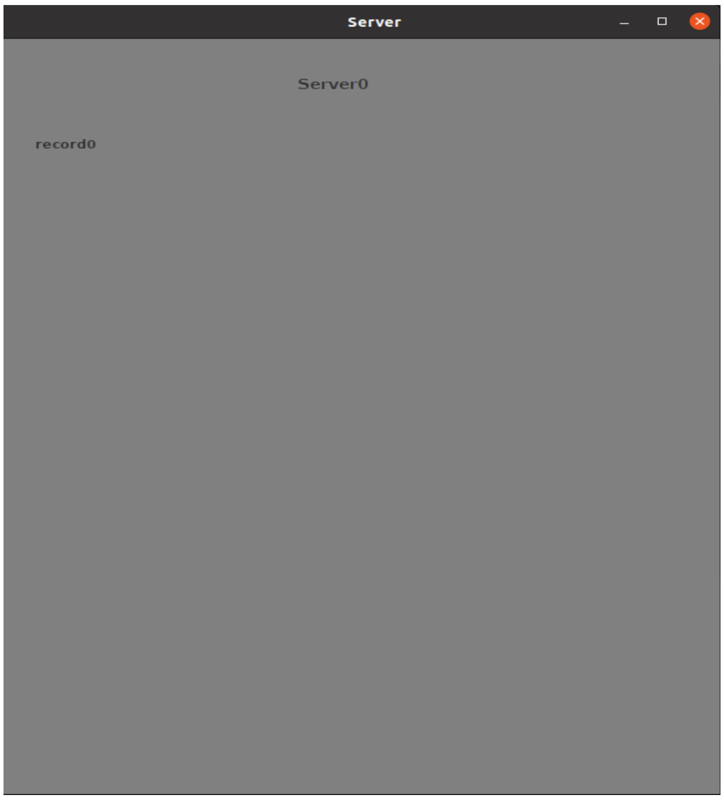


Figure 2.10 : Server GUI, after the append request of record0 is completed

2.6.3 Unit Tests

For correctness check, the author [2] runs some unit tests. In addition to the client and server code, a class that would act as a Byzantine server was implemented. This class was used to replace functional servers and test if the unbounded BDLO acted as it should. The author ran different configurations of the BDLO, in which some satisfied the condition $n \geq 3f + 1$, and some did not.

In all the configurations that the condition was met, all the Append and Get requests were completed as they should and all servers had the same sequence of records.

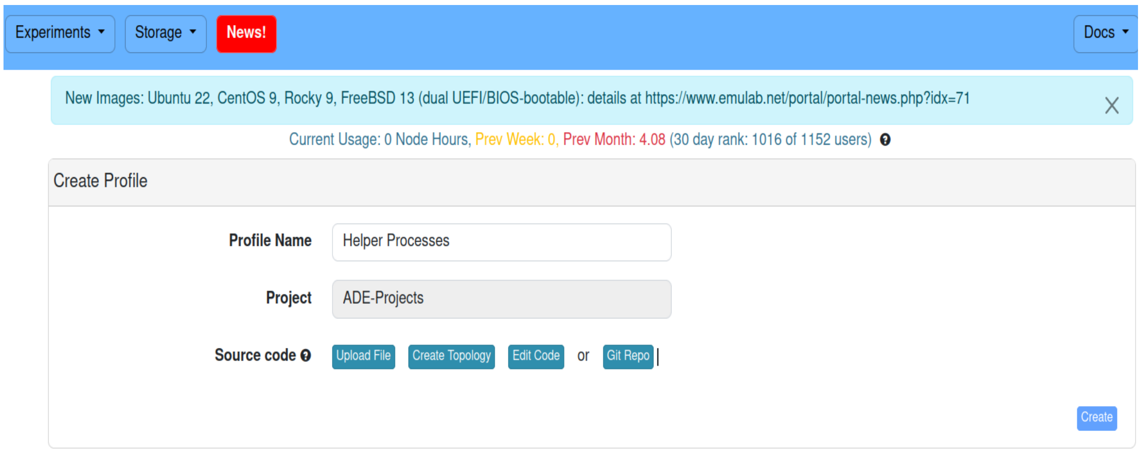
In all the configurations that the condition was not satisfied, the requests were not executed, which is what should happen.

For more details of the implementation, please check [2].

2.7 Emulab

In Section 5.4 we present an experimental evaluation of the Helper Processes algorithm. For the evaluation we use Emulab [6]. Emulab is a network testbed, which gives researchers a wide range of environments and hardware in which they can develop and evaluate their systems.

In order to test a system in Emulab, a user must first create an experiment profile. For the profile it is necessary for the user to give it a name, and also create a topology for the experiment. Figure 2.11 shows the user interface of the page that a user can create an experiment profile from.



New Images: Ubuntu 22, CentOS 9, Rocky 9, FreeBSD 13 (dual UEFI/BIOS-bootable): details at https://www.emulab.net/portal/portal-news.php?idx=71

Current Usage: 0 Node Hours, Prev Week: 0, Prev Month: 4.08 (30 day rank: 1016 of 1152 users)

Create Profile

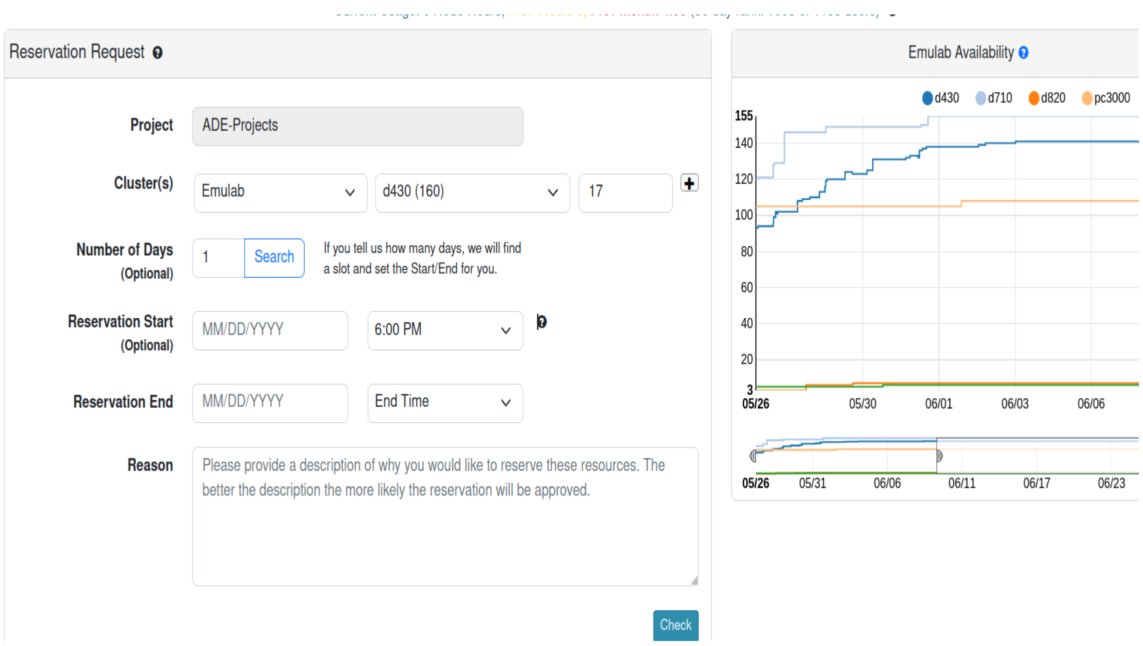
Profile Name	Helper Processes
Project	ADE-Projects
Source code	Upload File, Create Topology, Edit Code or Git Repo

Figure 2.11 : Create Experiment Profile in Emulab

In Figure 5.27 we present the topology that we use for the experimental evaluation of the Helper Processes algorithm. A topology is necessary in order to create the profile. In order to set up an experiment we need to request a reservation of the hardware needed for our experiment, at least one working day prior to the scheduled test (except if we request a small number of nodes). Figure 2.12 shows the user interface of the page where a client can reserve the nodes that are required for the experiment.

If the request is granted, then our set up is complete and we run our tests the day we scheduled the reservation.

To connect to a node that we reserved, we can use either the user interface of Emulab (terminal interface), or connect to it via SSH from our host machine. For the second option, we must provide Emulab with the RSA Public Key of our machine. We provide more information on our experimental evaluation in Section 5.4.


Reservation Request

Project	ADE-Projects		
Cluster(s)	Emulab	d430 (160)	17
Number of Days (Optional)	1	Search	If you tell us how many days, we will find a slot and set the Start/End for you.
Reservation Start (Optional)	MM/DD/YYYY	6:00 PM	
Reservation End	MM/DD/YYYY	End Time	
Reason	Please provide a description of why you would like to reserve these resources. The better the description the more likely the reservation will be approved.		

Check

Figure 2.13 : User Interface of Reservation Request page

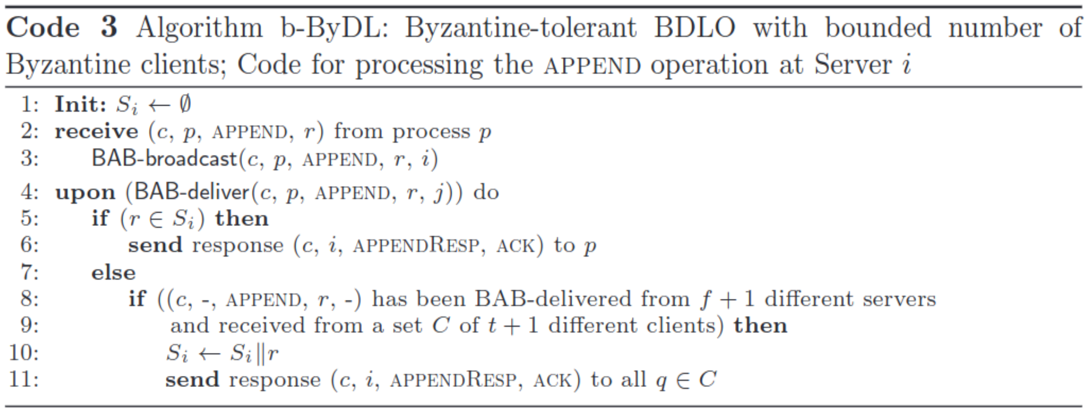
Chapter 3

Distributed Algorithms for Atomic Appends

As we have seen in Section 2.6.1, an implementation of what we call unbounded BDLO was developed. For the purposes of implementing solutions for the Atomic Appends problem, we need to also implement what we call *bounded BDLO* [3]. Therefore, we first present bounded BDLO and then we proceed to the Atomic Appends solutions.

3.1 Bounded Version of a BDLO

The term "bounded" refers to the fact that the fault tolerance of the system is limited, and cannot handle an unlimited number of malicious clients. A bounded BDLO is a specific version of a BDLO, which in order to append a Record r, it must receive a specific amount of append requests for the same record. More precisely, if the number of clients that can send an append request to the BDLO is n, at least $t + 1$ clients must send an Append Request for it, in order for it to be appended on the ledger. We denote t by the maximum number of clients that can be Byzantine, where $n \geq 3t + 1$. In this way, we ensure that the record, was not appended by accident or by a malicious client and the system remains, tolerant to Byzantine failures. This limitation is necessary to maintain the security and integrity of the ledger, and to prevent attacks or breaches.



Code 3 Algorithm b-ByDL: Byzantine-tolerant BDLO with bounded number of Byzantine clients; Code for processing the APPEND operation at Server i

```
 1: Init: $S_i \leftarrow \emptyset$
 2: receive ($c$, $p$, APPEND, $r$) from process $p$
 3:     BAB-broadcast($c$, $p$, APPEND, $r$, $i$)
 4: upon (BAB-deliver($c$, $p$, APPEND, $r$, $j$)) do
 5:     if ($r \in S_i$) then
 6:         send response ($c$, $i$, APPENDRESP, ACK) to $p$
 7:     else
 8:         if (($c$, -, APPEND, $r$, -) has been BAB-delivered from $f+1$ different servers
 9:             and received from a set $C$ of $t+1$ different clients) then
10:             $S_i \leftarrow S_i \| r$
11:             send response ($c$, $i$, APPENDRESP, ACK) to all $q \in C$
```

Figure 3.1: The algorithm of a server in a b-BDLO

Figure 3.1 shows the algorithm of a server that is part of a bounded BDLO. When it receives an append request for a record r, it first checks if the particular record has already been appended on the ledger. If it has then it sends back an Acknowledgement (line 5). If not, it stores it in a data structure, along with the amount of times the server received an Append Request for the particular record. The record is appended once at least $f + 1$ servers receive $t + 1$ append requests for the same record. The GET operation of a bounded BDLO server is the same with the unbounded version, which is shown in Figure 2.7. Additionally, the client code for the bounded BDLO is the same as in Figure 2.6. Later on, we modify the client code in order to solve the Atomic Appends problem, which we explain in the next section.

3.2 Algorithms for the Atomic Appends Problem

As we presented in Section 2.4, in order to solve the Atomic Appends problem we are in need of a middleman service. The objective of the middleman service is to check whether or not both clients have appended the records as they were supposed to. The records are appended onto their ledgers only when the middleman verifies that everything has been appended as agreed.

As we can see from from Figure 3.2, the clients send their records to the middleman service. After both records have been received, then the middleman service proceeds to append each record to the appropriate DLO. If a client doesn't send its record, then neither the other record is appended. Thus the Atomic Appends Problem, is resolved.

In the next sections we solve the Atomic Appends problem using two algorithms called Helper Processes and Smart BDLO. As we can see from from the Figure 3.2, the clients send their records to the middleman service. After both records have been received, then the middleman service proceeds to append each records to the appropriate Ledger. If a client doesn't send their record, then neither the other records are appended. Thus the Atomic Appends problem, is solved.

In the next sections we solve the Atomic Appends problem using two algorithms called Helper Processes and Smart BDLO [3]. In the first algorithm, the Byzantine-tolerant distributed middleman service is implemented using additional nodes we call "Helpers", while in the second algorithm the service is implemented using a special type of a BDLO, called *Smart BDLO*.

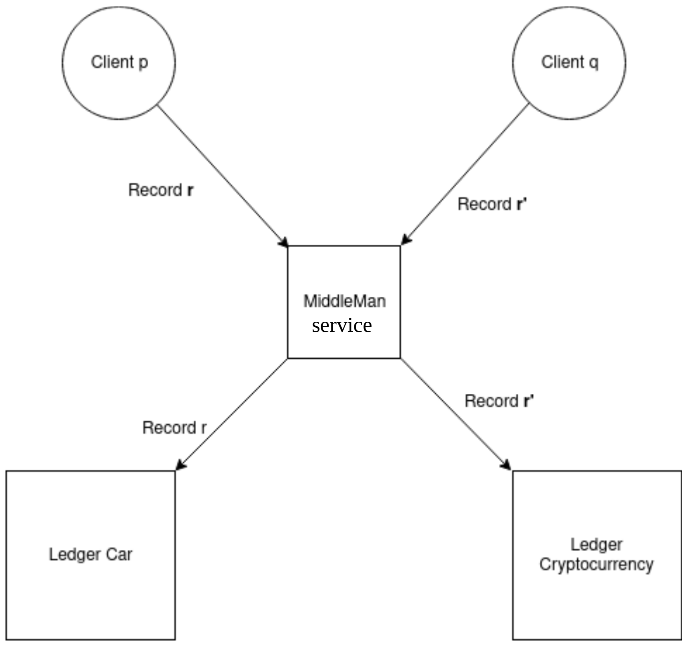


Figure 3.2 : Visualization of how a middleman would work

3.2.1 Helper Processes Algorithm

In this section we solve the Atomic Appends problem using the Helper Processes algorithm. In Figure 3.3 we can see a visual representation of how the algorithm works.

The helper processes are in reality clients of all the DLOs. Their purpose is to read periodically the Records of the unbounded BDLO, and search for matching records. When they discover a pair of matching records, they append each record of the pair to their respective bounded BDLO. Due to the reason that helper processes are a vital part of the system, we must bound the maximum number of them that could be Byzantine. So if the total number of helper processes is n, we say that there can at most t malicious, where $n \geq 3t + 1$. The helper processes algorithm is shown in Figure 3.4.

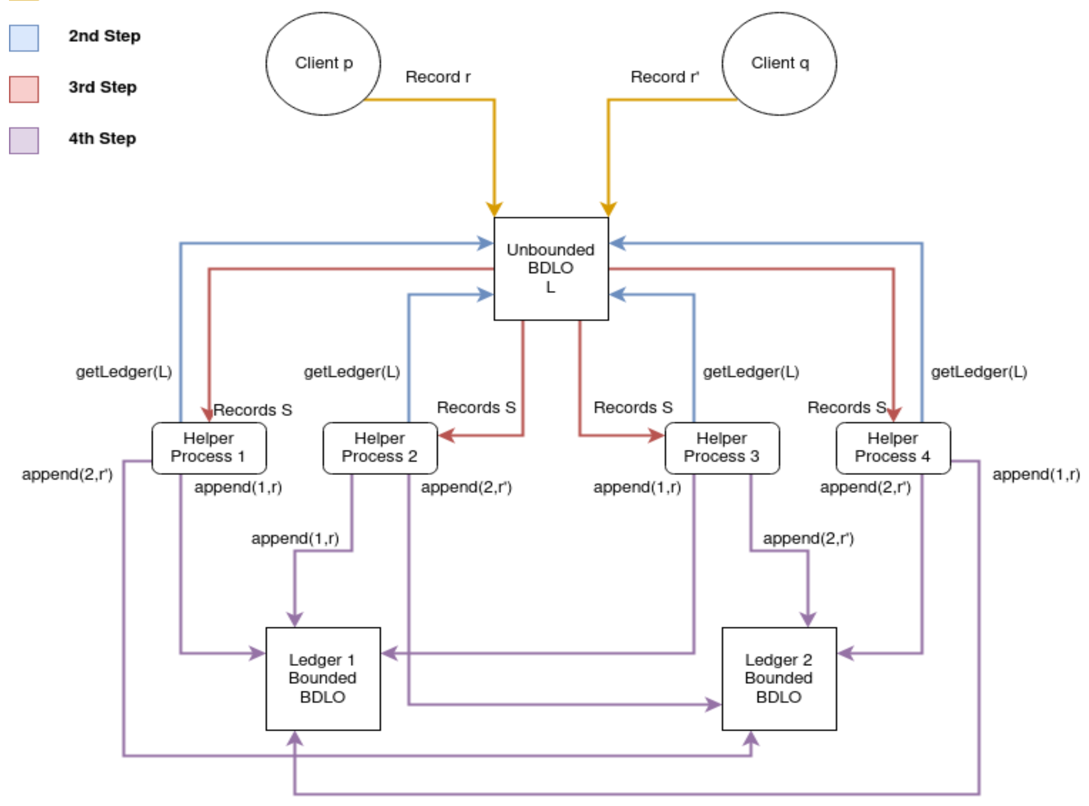


Figure 3.3: Visual Representation of Helper Processes Algorithm

Code 6 Algorithm used by a helper process to complete Atomic Appends operations; Code for process x

```
Init: $O_x \leftarrow \emptyset$
loop                                        ▷ Loop forever; execute loop body periodically
    $S_x \leftarrow \mathcal{L}.\text{GET}()$
    while $\exists r, r' \in S_x \setminus O_x : r.v = \langle p, \{p, q\}, r_p, \mathcal{L}_p, r_q \rangle \wedge r'.v = \langle q, \{p, q\}, r_q, \mathcal{L}_q, r_p \rangle$ do
        $\mathcal{L}_p.\text{APPEND}(r_p)$
        $\mathcal{L}_q.\text{APPEND}(r_q)$
        $O_x \leftarrow O_x \cup \{r, r'\}$
```

Figure 3.4 : The algorithm a helper process runs [3]

In Line 1, the O_x variable is used to store all the records that have already been appended onto their respective ledger, to avoid appending them again. Line 2 is the beginning of an infinite loop that is executed periodically. The first thing that a helper process does in the loop is to get the records from the unbounded BDLO, and store them into S_x. Subsequently, it iterates through the list, and once it finds a matching pair, then it appends each of them to their bounded BDLOs. Lastly the process adds the appended records in the list O_x . A matching pair r and r', have this value $r= \langle p, \{p, q\}, r_p, L_p, r_q\rangle$, and $r' = \langle q, \{p, q\}, r_q, L_q, r_p\rangle$. The first element in the value of a record is the ID of the client who appended the record (in r.v is client p). The second element is a set of IDs of the clients that have come into an agreement (in r.v this set is p and q). The third element is the value that client p wants to append and the fourth on which ledger the record must be appended. The fifth element is the value of what client q is expected to append, and the last element indicates on which ledger the record of client q is expected to be appended.

3.2.2 Algorithm using Smart BDLO

The Smart BDLO algorithm is very similar to the previous one. The main difference is that the work of the Helper Processes is being done within the unbounded BDLO, so the extra instances that are used in the Helper Processes algorithm are not needed in this one. Each server in the collection of the BDLO also acts as a clients for the bounded BDLOs. We can see the visualization of the algorithm in Figure 3.5.

As we can see, the Smart BDLO consists of *n* servers. The servers work exactly as they would in the Helper Processes algorithm, but with the addition of searching matching records, and appending them on their respective ledgers. We can see the Append operation they run in Figure 3.6 (GET operation can same as Figure 2.7). Lines 5-10 represent the work that the Helper Processes were responsible for in the previous implementation.

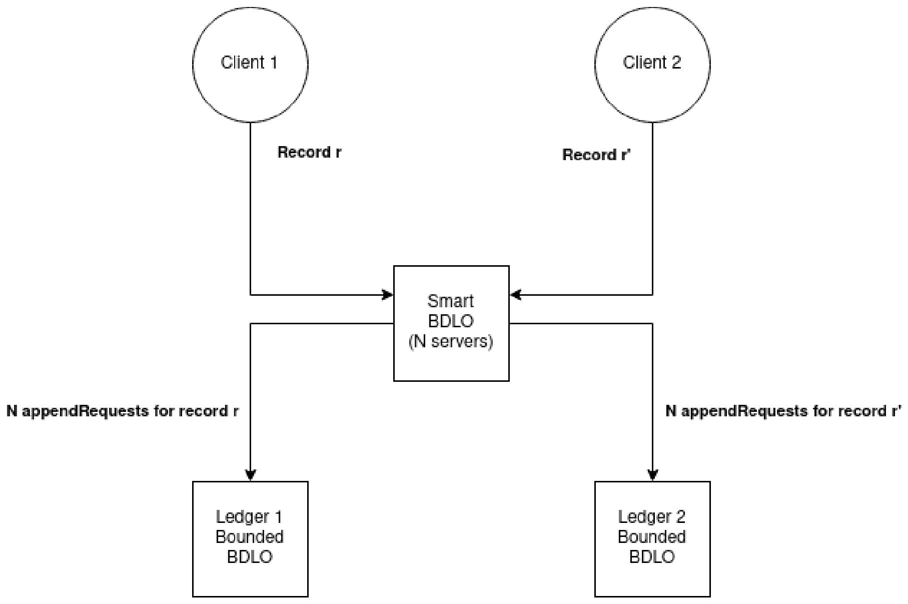
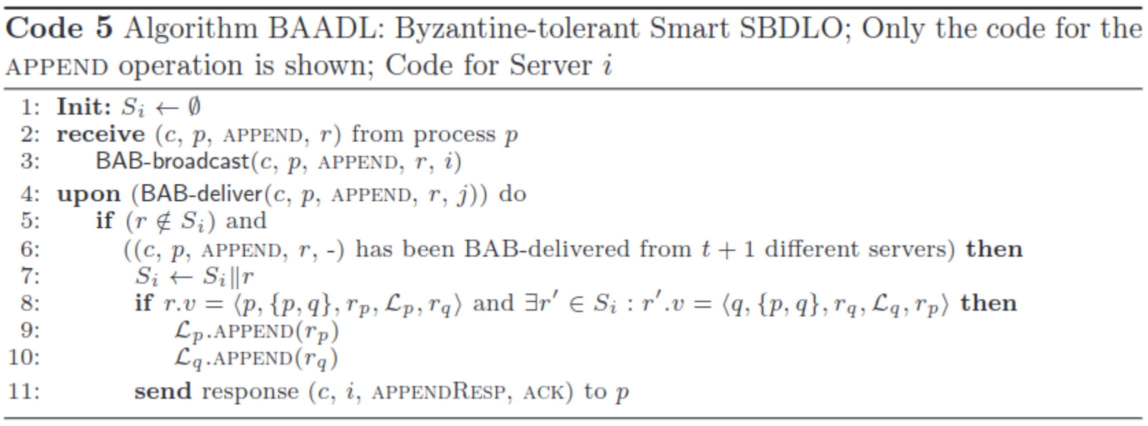


Figure 3.5: Visual representation of the Smart-BDLO algorithm



Code 5 Algorithm BAADL: Byzantine-tolerant Smart SBDLO; Only the code for the APPEND operation is shown; Code for Server i

```
Init: $S_i \leftarrow \emptyset$
receive $(c, p, \text{APPEND}, r)$ from process $p$
    BAB-broadcast$(c, p, \text{APPEND}, r, i)$
upon (BAB-deliver$(c, p, \text{APPEND}, r, j)$) do
    if $(r \notin S_i)$ and
      $((c, p, \text{APPEND}, r, -)$ has been BAB-delivered from $t+1$ different servers) then
        $S_i \leftarrow S_i \| r$
        if $r.v = \langle p, \{p, q\}, r_p, \mathcal{L}_p, r_q \rangle$ and $\exists r' \in S_i : r'.v = \langle q, \{p, q\}, r_q, \mathcal{L}_q, r_p \rangle$ then
            $\mathcal{L}_p.\text{APPEND}(r_p)$
            $\mathcal{L}_q.\text{APPEND}(r_q)$
        send response $(c, i, \text{APPENDRESP}, \text{ACK})$ to $p$
```

Figure 3.6: Algorithm for Smart-BDLO (only append operation is shown) [3]

3.3 Use Case Scenarios for Both Algorithms

From the previous sections, its very clear that the Smart BDLO algorithm needs less instances (servers) to run. In Section 5.4 we prove that the Helper Processes algorithm is significantly slower than the Smart BDLO algorithm due to the GET requests a Helper Process needs to execute.

So if the main priority of a system is performance and the use of a small number of servers then the Smart BDLO is perfect for this scenario. However, we cannot use the Smart BDLO algorithm when the unbounded BDLO is a black box, since there is no way for us to modify the server code to work as we want to. In this case the most suitable algorithm would be the Helper Processes algorithm. A helper process would just need to execute a GET request to read the Records of the unbounded BDLO, and execute APPEND requests once it finds a matching pair. Therefore both algorithms can be used in different projects with different criteria.

Chapter 4

Implementation

In this chapter we present the methods and classes we developed. First we display the Record class, which has similar structure to the Records that appear in the Figures 3.5 and 3.8. Then we continue with the implementation of the bounded BDLO, which as we discussed before, it is a necessary component for the implementation of both Helper Processes and Smart BDLO algorithms. In the next two sections we carry on with showcasing the code that we developed in order to implement the Helper Processes and Smart BDLO algorithm. Lastly we present some modifications that we made, in order for an instance to be able to communicate with more than one ledger.

4.1 Record Class

The Record class is used to create objects that have similar structure to the Records that we have seen in the pseudo-codes. A Record object has the following variables :

1. String Client1 – The ID of the client that sent the request
2. String Client2 – The ID of the client that came to an agreement with Client1
3. String Value1 – The value that Client1 wants to append
4. String Value2 – The value that Client2 is expected to append
5. String Ledger1 – On which ledger Client1 will append to

6. String Ledger2 – On which ledger Client2 is expected to append to

Additionally we also created some methods that would help us implement the algorithms.

The first one is called createMatchingRecord, which returns another Record. That record is what a helper process looks for in the ledger in order to execute append Requests to the bounded BDLOs. In Figure 4.1 we see the code, which is simply just swapping the variables of client1 with those of client2 for the given Record r.

```
public Record createMatchingRecord(Record r) {
        return new Record(r.getClient2(), r.getClient1(),r.getValue2(), r.getValue1(), r.getLedger2(),
r.getLedger1());
    }
```

Figure 4.1 : creatingMatchingRecord method

Another function we implemented was toString which is used for the user interface. We can see the code in Figure 4.2.

```
public String toString(){
        return ("Client1: " + this.getClient1() + ", Client2: " + this.getClient2() + ", Value1: " +
this.getValue1() + ", Value2: " + this.getValue2() + ", Ledger1: " + this.getLedger1() + ", Ledger2: " +
this.getLedger2());
    }
```

Figure 4.2 : toString method

Also we implemented the equals function, which is used to compare records. Without this function, there could be instances, where two Record objects with the same values, do not appear as equal.

```
public boolean equals(Object value) {
        if (this == value) return true;
        if (value == null || getClass() != value.getClass()) return false;
        Record v = (Record) value;
        return this.value1.equals(v.getValue1()) &&
            this.value2.equals(v.getValue2()) &&
            this.getLedger1().equals(v.getLedger1()) &&
            this.getLedger2().equals(v.getLedger2()) &&
            this.getClient1().equals(v.getClient1()) &&
            this.getClient2().equals(v.getClient2());
    }
```

Figure 4.3 : equals method

The final important method we implemented was the function hashCode. The creation of this method was necessary because we used the class HashMaps in order to store records in them. If this method was not implemented, then the contains method of hashMap would not have worked correctly. It is crucial for the contains method to work as it is supposed to, because we use it in order to search for matching Records within the HashMap variable. Before implementing the code presented in Figure 4.4, we tested the Record class by adding a Record r in a HashMap. Then used another Record r' which had the same values as r, but the contains method returned False when we tried to search for r with r'. This happens because the default hashCode method uses the memory location of an object instead of its value to generate a Hash.

```
public int hashCode() {
      return this.toString().hashCode();
  }
```

Figure 4.4: hashCode method

4.2 Bounded-BDLO Implementation

In this section we present the modifications that were made in order for a server to append a request on its own ledger, when it only receives t + 1 same requests. For this implementation, we had to add a configuration setting in the system.config file. That setting is called *system.clients.t* which allows the servers to know the maximum number of clients that can be Byzantine.

For the implementation, three new variables were necessary to be introduced. In Figure 4.5 we can see these variables.

```
private Map<Record,Integer> appendRequests;
private int t;
private List<Record> LedgerFinal;
```

Figure 4.5: New variables created to implement bounded-BDLO

The variable appendRequests(HashMap) is used to store the Records that have been received so far along with a counter for each one. The key is the Record that was sent in the append Request, and the value is an integer which indicates how many times this record was received. Furthermore the variable t is the maximum number of clients that can be Byzantine. As mentioned before, the value is read from the system configuration. The last variable is a List that stores all the Records that have successfully been appended onto the ledger.

A bounded Server works the same way as an unbounded one, except when it receives an append request. The method that had to be modified is called appExecuteOrdered, and these are the case scenarios.

- *The server receives an append request for the record r, for the first time. (Figure 4.6)*
- *The server receives an append request for the record r, for k^{th} time, where $k \leq t$. (Figure 4.7)*
- *The server receives an append request for the record r, t + 1 times. (Figure 4.8)*
- *The server receives an append request for the record r, for n^{th} time, where $n > t + 1$. (Figure 4.9)*

```
1  case APPENDLEDGER:
2
3          Record key1 = (Record)objIn.readObject();
4          System.out.println("The APPEND message with C="+key1+" has been delivered");
5          value = (V)objIn.readObject();
6
7          if (!appendRequests.containsKey(key1)){
8                  appendRequests.put((Record)key1, 1);
9                  System.out.println("New record inserted in appendsRequest with key "+ key1);
10                 hasReply = true;
11                 objOut.writeUTF("RECEIVED THE FIRST APPEND REQUEST FOR RECORD "+ key1);
12         }
```

Figure 4.6 : The server receives the record r for the first time

In Figure 4.6, we see that when a Record r is appended for the first time, it is inserted in the appendRequests HashMap(line 8) with value 1. The client should receive the message seen in line 11.

```
1  else if (appendRequests.get(key1) < t){
2           int old_value = appendRequests.get(key1);
3           appendRequests.replace((Record)key1,(Integer)(old_value) , (Integer)(old_value+1));
4           System.out.println("Received another record for key "+key1+ "and now it has "+
appendRequest.get(key1));
5           hasReply = true;
6           objOut.writeUTF("RECEIVED ANOTHER APPEND REQUEST FOR RECORD " + key1);
7
8  }
```

Figure 4.7 : The server receives another request for record r, but has not reached t+1 yet.

As shown in Figure 4.7, when the server receives another Record r for the kth time, where k < t+1 then it replaces the value of the previous record by adding 1 to the old value (lines 2 and 3). The client who sends the kth request receives the message seen in line 6.

```
1  else if (appendRequests.get(key1) == t) {
2           System.out.println("The append request for the record "+ key1 + " has been received from at
least t+1 clients. It has now been appended onto the ledger");
3           LedgerFinal.add((Record)key1);
4           int old_value = appendRequests.get(key1);
5           appendRequests.replace((Record)key1,(Integer)(old_value) , (Integer)(old_value+1));
6           System.out.println(LedgerFinal.size());
7           objOut.writeUTF("THE RECORD " + key1 + " HAS BEEN APPENDED ONTO THE LEDGER");
8           hasReply = true;
9  }
```

Figure 4.8: The server receives t+1 append requests for the record r

In Figure 4.8, we can see the code for when the server receives total of $t + 1$ append requests for the Record r. Once it is received, it is appended onto the server's ledger (line 3), the value of the record in the HashMap its increased by one (line 4 and 5), and the client who sent the t+1th request receives the message seen in line 7.

```
1  else {
2           objOut.writeUTF("THE RECORD " + key1 + " HAS ALREADY BEEN APPENDED ON THE
LEDGER");
3           hasReply = true;
4  }
```

Figure 4.9 : The server receives another request, for an already appended record

Finally, Figure 4.9 shows when a client sends an append request for an already appended record it gets the message seen in line 2.

4.3 Helper Processes Implementation

The focus of this section is to present the implementation of the Helper Processes algorithm that we saw in Section 3.2.1. A helper process, as we mentioned before, is essentially a client which reads the records of the unbounded BDLO, and when it finds two matching records, it sends an append request to the appropriate ledgers.
To implement the helper process, we modified the client class that was created in the previous thesis. In addition, for simplicity purposes, we assumed there were only two bounded BDLOs. In reality there could be many more, and it would be fairly easy to implement with more than two bounded BDLOs. The name for the bounded BDLOs is "1" and "2", respectively.

In Figure 4.10 we can observe the most important method of a helper process called readLedger. Firstly a HashMap named alreadySent is initialized, before we enter the infinite loop. The variable as the name suggests, is used to store all the records for which the process has already sent an append request to the ledger it belongs.
The infinite loop is executed every 10 seconds using the sleep method (line 42). At the beginning of the loop, the process sends getRequest to the unbouded-BDLO (line 4), and gets back a list of records. For complexity reasons, we decided to convert the list to a HashMap (line 7), in order to access the records quicker. Afterwards we remove all the records from the Map that are stored in the variable alreadySent, using the method removAll(). The next step, is to iterate through the hashMap. For every record, it creates the matching record, using the method we saw in Figure 4.1, so if that record also exists in the map, then it should execute the appendsRequest (line 18). Since the request has been sent, then the particular record is stored in the alreadySent map (line 30).

```
public void readLedger(){
        Map<Record, Integer> alreadySent = new HashMap<>();
        while(true){
        List<Record> list =  GetLedger(calculateKey());
        if (list.size() > 0 ){

        map = list.stream().collect(Collectors.toMap(s -> s, s -> list.indexOf(s)));
        map.keySet().removeAll(alreadySent.keySet());
        map.forEach((key, value) -> {
        Record temp = key.createMatchingRecord(key);
         if(alreadySent.containsKey(key)){
            System.out.println("Already Sent");
        }
        else if (map.containsKey(temp)){
            System.out.println("Found Matching Records.");
            System.out.println("Executing Atomic Appends");
            try {
                System.out.println(AppendLedger((K) key, (V) value));
            } catch (ClassNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (ExecutionException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

            alreadySent.put(key, value);
            }

        });

        map.clear();

        }
        else {
            System.out.println("Nothing in Ledger yet");
        }
        try {
            sleep(10000);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    }
```

Figure 4.10 : Helper Process, searching and appending matching records

A helper process sends an appendRequest using the method AppendLedger(Record, Value). This method has been modified in order, to send the request to the correct ledger. We can observe the code for the method, in the Figure 4.11.

```
1  public String AppendLedger(K key, V value) throws ClassNotFoundException, InterruptedException, ExecutionException {
2          try (ByteArrayOutputStream byteOut = new ByteArrayOutputStream();
3               ObjectOutput objOut = new ObjectOutputStream(byteOut);) {
4
5                          objOut.writeObject(BRequestType.APPENDLEDGER);
6                          objOut.writeObject(key);
7                          objOut.writeObject(value);
8
9                          objOut.flush();
10                         byteOut.flush();
11                         System.out.println("The message APPEND with C=" + key + " has been broadcasted to the servers of the Ledger");
12
13                         byte[] reply;
14                         Record temp = (Record) (key);
15                         if (temp.getLedger1().equals("1")) {
16                                 reply = serviceProxy1.invokeOrdered(byteOut.toByteArray());
17                         } else if (temp.getLedger1().equals("2")){
18                                 reply = serviceProxy2.invokeOrdered(byteOut.toByteArray());
19                         }
20                         System.out.println("The same reply from at least f+1 different servers for the " + key + " request have just arrived");
21                         try {
22                             if (reply.length == 0) {
23                                 return null;
24                             }
25
26                             else {
27                                 try (ByteArrayInputStream byteIn = new ByteArrayInputStream(reply);
28                                 ObjectInput objIn = new ObjectInputStream(byteIn)) {
29                                 return objIn.readUTF();
30                                 }
31
32                             }
33                         } catch (NullPointerException e) {
34                                 System.out.println("NULL POINTER EXCEPTION");
35                         }
36
37                         } catch (IOException e) {
38                                 System.out.println();
39                         }
40                         return null;
41         }
```

Figure 4.11 : AppendLedger method

In lines 16 and 18 is where the APPEND request is sent. A service proxy is a class that comes with the BFT-Smart library, and is used to communicate with the BDLOs. The serviceProxy1 is responsible for the communication for the bounded BDLO with ID 1, and serviceProxy2 for the bounded BDLO with ID 2. Once those BDLOs receive t + 1 append requests for the same record, then they append it onto their ledger.

The result of the request is printed out to the helper process's terminal.

4.4 Smart-BDLO Implementation

A Smart-BDLO is essentially an unbounded-BDLO which has the capability to do the work of a helper processes instantly. So it works as a server and a client at the same time. More specifically the servers of the Smart-BDLO check their ledgers for matching records, and if there is a matching pair, then they execute append requests to the bounded BDLOs. A helper process would have to request the ledger from the unbounded servers, but since now the servers do all the work, that overhead is now gone.

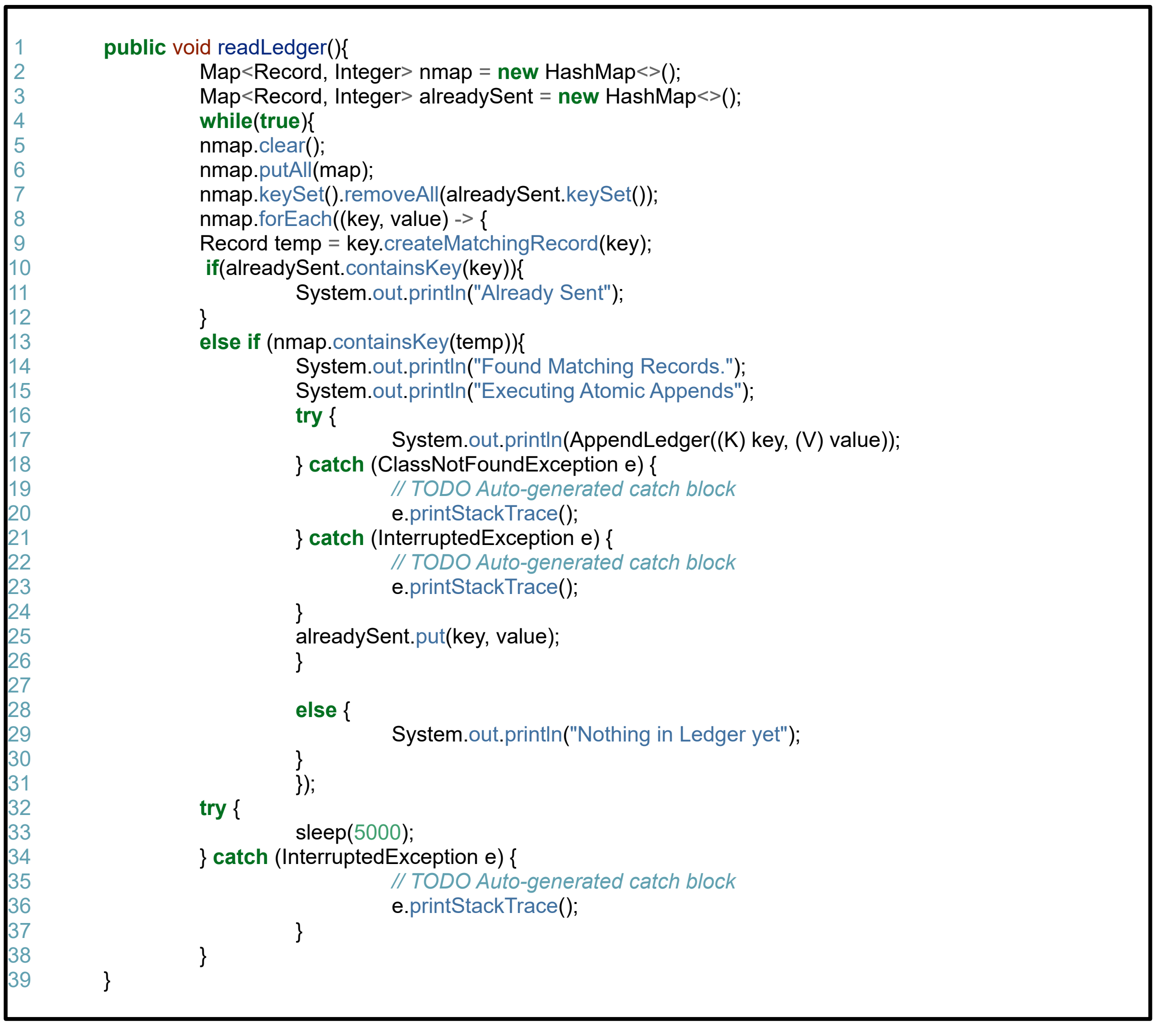
For the implementation we used the library Runnable in order to use multithreading. The extra thread is responsible for checking the ledger for matching pairs, and also for the append requests.

Also for complexity reasons, an addition had to be made for when the servers received an append request. Until now, a record was put only in a list, now we also append it on a HashMap with the name map, which allows us to search for matching records quicker (O(1) specifically).

In Figure 4.12 we see the method readLedger, which is the code that the extra thread is executing. This method is almost the same with the code seen in Figure 4.10. We use a new HashMap called nmap, which in every iteration gets all the data that are currently stored in the variable map. The main reason we do this is to avoid the scenario while we iterate through the map, a record is appended onto it. If this happens, an error would occur when the thread iterates though the map.

Additionally, the servers must be able to send append requests as a client. So we add the method AppendLedger, as seen in the Figure 4.11, on the server class. When a server finds a matching pair of Records, as presented in Figure 4.12 (lines 13), it calls the method AppendLedger (line 17) which sends an append request to the appropriate bounded BDLO. The result of the request is printed out to the server's terminal.

For a further understanding of the algorithm we can see the diagram presented in in Figure 4.13.



```
public void readLedger(){
    Map<Record, Integer> nmap = new HashMap<>();
    Map<Record, Integer> alreadySent = new HashMap<>();
    while(true){
    nmap.clear();
    nmap.putAll(map);
    nmap.keySet().removeAll(alreadySent.keySet());
    nmap.forEach((key, value) -> {
    Record temp = key.createMatchingRecord(key);
     if(alreadySent.containsKey(key)){
        System.out.println("Already Sent");
    }
    else if (nmap.containsKey(temp)){
        System.out.println("Found Matching Records.");
        System.out.println("Executing Atomic Appends");
        try {
            System.out.println(AppendLedger((K) key, (V) value));
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        alreadySent.put(key, value);
        }

        else {
            System.out.println("Nothing in Ledger yet");
        }
        });
    try {
        sleep(5000);
    } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

Figure 4.12 : readLedger method on a Smart-BDLO server

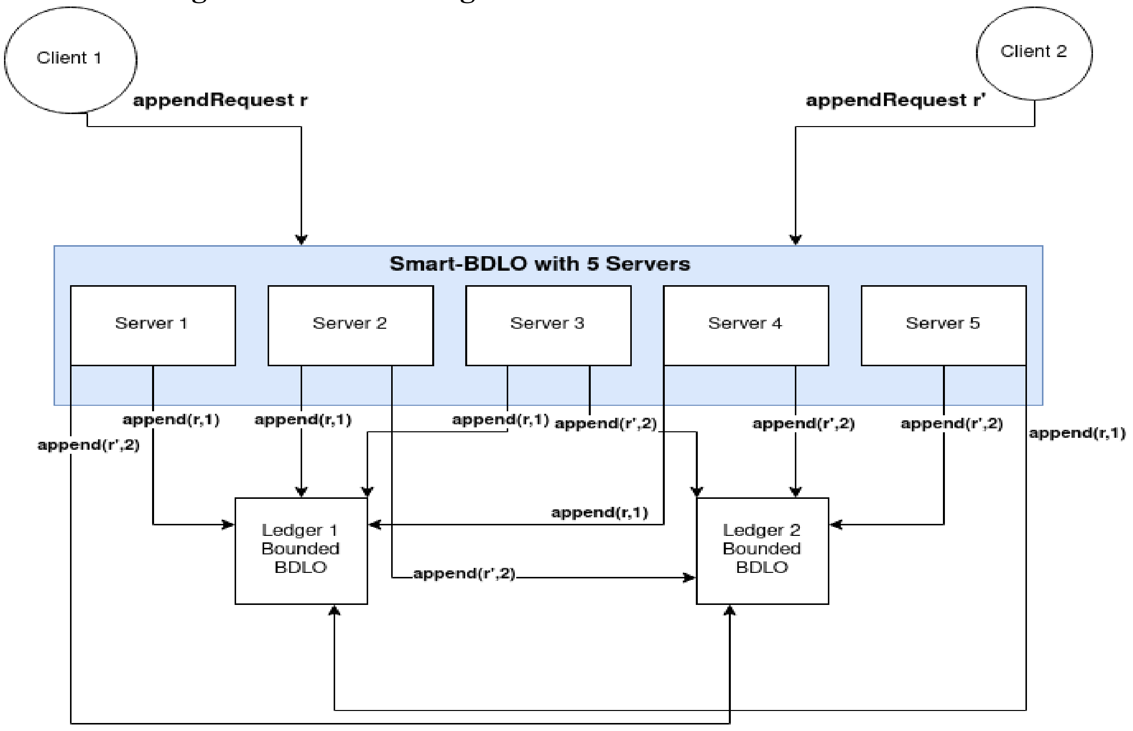


Figure 4.13 : Smart-BDLO architecture

4.5 Necessary Modifications

As it was previously implemented in [2], a node could only access one ledger while running. But for our algorithms to work, a node must be able to have access to multiple ledgers at a time. The way a node was connecting to a ledger, was with the class ServiceProxy, which was implemented by the BFT-Smart library. The serviceProxy would read the servers from hosts.config (Figure 2.3), by default, and connect to the ledger that consists of those servers. So in order to solve this, we decided that it would be best to add a function in serviceProxy, which would allow us to connect to a ledger, by reading a different configuration file other than hosts.config. For example, a helper process would now read from 3 different configuration files. The first one would be the file for the unbounded BDLO, the second one would be for the bounded BDLO for the ledger with ID 1, and the last one for the bounded BDLO for the ledger with ID 2. Figure 4.14 shows the two methods that were added in the serviceProxy class.

```
public ServiceProxy(String filename, int processId, String viewName) {
        this(processId, null, null, null, null, filename, viewName);
}


public ServiceProxy(int processId, String configHome, Comparator<byte[]> replyComparator, Extractor
replyExtractor, KeyLoader loader, String filename, String viewName) {


                if (configHome == null) {
                        init(processId, loader, filename, viewName);
                } else {
                        init(processId, configHome, loader);
                }

                replies = new TOMMessage[getViewManager().getCurrentViewN()];

                comparator = (replyComparator != null) ? replyComparator : new Comparator<byte[]>() {
                        @Override
                        public int compare(byte[] o1, byte[] o2) {
                                return Arrays.equals(o1, o2) ? 0 : -1;
                        }
                };

                extractor = (replyExtractor != null) ? replyExtractor : new Extractor() {

                        @Override
                        public TOMMessage extractResponse(TOMMessage[] replies, int sameContent, int
lastReceived) {
                                return replies[lastReceived];
                        }
                };
        }
```

Figure 4.14 : Added constructors in Service Proxy class

We use the property of Java, called overloading, which allows us to have methods with the same name but different parameters. In Figure 4.14, we added two constructor methods, which allow us to enter a a name of a configuration file, that includes the servers of a ledger.

The serviceProxy extends the class TOMSender, and as seen in line 10, the constructor of the second is called. So we also had to implement a method for that class as well.

```
1  public void init(int processId, KeyLoader loader, String filename, String viewName) {
2                          this.viewController = new ClientViewController(processId, loader, filename, viewName);
3                          startsCS(processId);
4          }
```

Figure 4.15 : Added constructor in TOMSender class

In line 2 of Figure 4.15 is where the TOMSender creates the view of the ledger(which servers the ledger consists of) using the ClientViewController class. For the ClientViewController, to be able to create the View, we must also manipulate the following classes:

- ViewController (Extended by ClientViewController)
- TOMConfiguration (Used by ViewController)
- Configuration (Extended by TOMConfiguration)
- HostsConfig (Used by Configuration)

The following figures show the added code for each one of these classes.

```
public ViewController(int procId, KeyLoader loader, String filename) {
      this.staticConf = new TOMConfiguration(procId, loader,filename);
  }
```

Figure 4.16 : Added Code for ViewController

```
public TOMConfiguration(int processId, KeyLoader loader, String filename) {
      super(processId, loader, filename);
  }
```

Figure 4.17 : Added Code for TOMConfiguration

```
public Configuration(int procId, KeyLoader loader, String filename) {
                    logger = LoggerFactory.getLogger(this.getClass());
                    processId = procId;
                    keyLoader = loader;
                    init(filename);
}

protected void init(String filename) {
                    logger = LoggerFactory.getLogger(this.getClass());
                    String filename1 = new String(filename);
                    try {
                                hosts = new HostsConfig(configHome, filename);

                                loadConfig();
                    ...// THERE IS MORE CODE. THIS IS THE ONLY PART THAT I CHANGED

 }

```

Figure 4.18 : Added Code for Configuration

```
public HostsConfig(String configHome, String fileName) {
      loadConfig(configHome, fileName);
   }

private void loadConfig(String configHome, String fileName){
      try{
         String path =  "";
         String sep = System.getProperty("file.separator");
         if(configHome.equals("")){
               if (fileName.equals(""))
                  path = "config"+sep+"hosts.config";
               else
                  path = "config"+sep+fileName;
         }else{
               if (fileName.equals(""))
                  path = configHome+sep+"hosts.config";
               else
                  path = configHome+sep+fileName;
         }
         FileReader fr = new FileReader(path);
         BufferedReader rd = new BufferedReader(fr);
         String line = null;
         while((line = rd.readLine()) != null){
            if(!line.startsWith("#")){
               StringTokenizer str = new StringTokenizer(line," ");
               if(str.countTokens() == 4){
                  int id = Integer.valueOf(str.nextToken());
                  String host = str.nextToken();
                  int port = Integer.valueOf(str.nextToken());
                  int portRR = Integer.valueOf(str.nextToken());
                  this.servers.put(id, new Config(id, host, port, portRR));
               }
            }
         }
         fr.close();
         rd.close();
      }catch(Exception e){
         LoggerFactory.getLogger(this.getClass()).error("Could not load configuration file",e);
      }
   }

```

Figure 4.19 : Added Code for HostsConfig

Figure 4.19 shows how a helper process would connect to all 3 ledgers that needs to connect.

```
serviceProxy = new ServiceProxy("unbounded.config", clientId, "unbounded"); // connects to the Unbounded Ledger
serviceProxy1 = new ServiceProxy("ledger1.config", clientId, "ledger1"); // connects to Ledger with ID 1
serviceProxy2 = new ServiceProxy("ledger2.config", clientId, "ledger2");
```

Figure 4.20 : Initialisation for ServiceProxy variables in a Helper Process

Also, now the original clients can now send Get requests to the bounded BDLOs. Previously the could get the ledger only from the unbounded.

Chapter 5

Visual Representation of the Algorithms

In this chapter we present a guide on how to prepare a system to run our prototype. We give a visual representation of both Helper Processes and Smart BDLO algorithms, along with instructions on how to launch each one. At the end of the chapter we compare the performance of the algorithms using an experiment which we executed on Emulab [6].

5.1 Prerequisites

In preparation for running the algorithms, we must do a couple of steps. The first step is to install Java Runtime Environment 1.8 or greater, as it is required from the BFT-Smart library.

Install JRE 1.8 on **Debian** Systems:

1. Open terminal
2. Execute the command **sudo apt-get update**
3. Execute the command **sudo apt-get install openjdk-8-jdk**

Install JRE 1.8 on **Windows** Systems:

1. Open this link
2. Accept the Licence Agreement
3. Download the file
4. Install the executable

The second step is risky and should be avoided in any system that is in production. We only do this, because the programs are only prototypes, and they are used just for demonstration. That being said, the second step is to enable old protocols that may be vulnerable to malicious attacks, but they are used in our system which doesn't launch if we don't do this step. This was also a problem in the previous study [2]. Ideally we would like to find a safer method to bypass this problem. However the handling of network communication is implemented within the BFT-Smart library, and it would take a lot of effort and time to manipulate the library in order to work as we want to.

Steps on **Debian** Systems:

1. Locate the Java Security file (usually in /usr/lib/jvm/java-8-openjdk/conf/security/)
2. Use a text editor to open it (for example nano java.security)
3. Find the following line

```
jdk.tls.disabledAlgorithms=SSLv3, TLSv1, TLSv1.1, RC4, DES, MD5withRSA,
    DH keySize < 1024, EC keySize < 224, 3DES_EDE_CBC, anon, NULL,
    include jdk.disabled.namedCurves

```

4. Comment out the line
5. Save the file

Steps on **Windows** Systems:

1. Locate the Java Security file (usually in `C:\Program Files\Java\jdk1.8\jre\lib\security`)
2. Open the java.security file
3. Find the following line

```
jdk.tls.disabledAlgorithms=SSLv3, TLSv1, TLSv1.1, RC4, DES, MD5withRSA,
    DH keySize < 1024, EC keySize < 224, 3DES_EDE_CBC, anon, NULL,
    include jdk.disabled.namedCurves

```

4. Comment out the line

5. Save the file

After these steps we are ready to launch the implementation of the algorithms.

5.2 Helper Processes Execution

In the section, we demonstrate the Helper Process implementation, but first we present how configure and launch each instance.

5.2.1 Instructions on How to Launch the System

For our scenario, we have 4 servers on each ledger(unbounded-BDLO, bounded-BDLO 1, bounded-BDLO 2) and 4 helper processes. We can launch each instance on a different system which its IP is accessible to the other systems, or we can launch all the instances on the same system.

Firstly we must configure the servers as seen in Section 2.5.2. For the BDLOs we only need to configure the hosts and system configuration files. These can be found in HelperProcessesImplementation/(ledger 1, ledger2, unbounded Version)/library/config.

Lastly we need to configure the helper processes and the clients. We need to configure the following files, unbounded.config, ledger1.config, ledger2.config(IP Addresses to connect to the ledgers). The files can be found in HelperProcessesImplementation /helperProcesses/library/config and HelperProcessesImplementation/clients/library/config.

Now that we have configured everything, we instruct on how to launch each instance. The instances must launch in this order:

1. bounded BDLOs
2. unbounded BDLO
3. Helper Processes
4. Clients

In order to launch the bounded BDLOs, we must go to HelperProcessesImplementation/ledger1/library and HelperProcessesImplementation /ledger2/library, open a terminal and execute these commands.

```
./runscripts/smartrun.sh bftsmart/demo/blockchain/BServer 0 <IP of the Server>
./runscripts/smartrun.sh bftsmart/demo/blockchain/BServer 1 <IP of the Server>
./runscripts/smartrun.sh bftsmart/demo/blockchain/BServer 2 <IP of the Server>
./runscripts/smartrun.sh bftsmart/demo/blockchain/BServer 3 <IP of the Server>
```

We do the same for the unbounded BDLO, but in this directory HelperProcessesImplementation/unboundedVersion/library.

Now we are going to launch Helper Processes. We must go to the directory HelperProcessesImplementation/helperProcesses/library, open a terminal and execute the following commands.

```
./runscripts/smartrun.sh bftsmart/demo/blockchain/HelperProcess 0
./runscripts/smartrun.sh bftsmart/demo/blockchain/HelperProcess 1
./runscripts/smartrun.sh bftsmart/demo/blockchain/HelperProcess 2
./runscripts/smartrun.sh bftsmart/demo/blockchain/HelperProcess 3
```

And lastly we launch the clients. Keep in mind that the IDs of the clients must start from the last ID of a Helper Process, otherwise the communication between the clients and the servers will be problematic. So in this case, the first client will have ID 4. In this scenario we will have just 2 clients. Launch the clients by opening a terminal in the directory HelperProcessesImplementation /clients/library and executing the following commands.

```
./runscripts/smartrun.sh bftsmart/demo/blockchain/ClientGui 4
./runscripts/smartrun.sh bftsmart/demo/blockchain/ClientGui 5
```

5.2.2 Demonstration of Helper Processes Implementation

First we launch the bounded BDLOs. Both of them must have 4 instances as seen in Figure 5.1 (only this first server is shown).

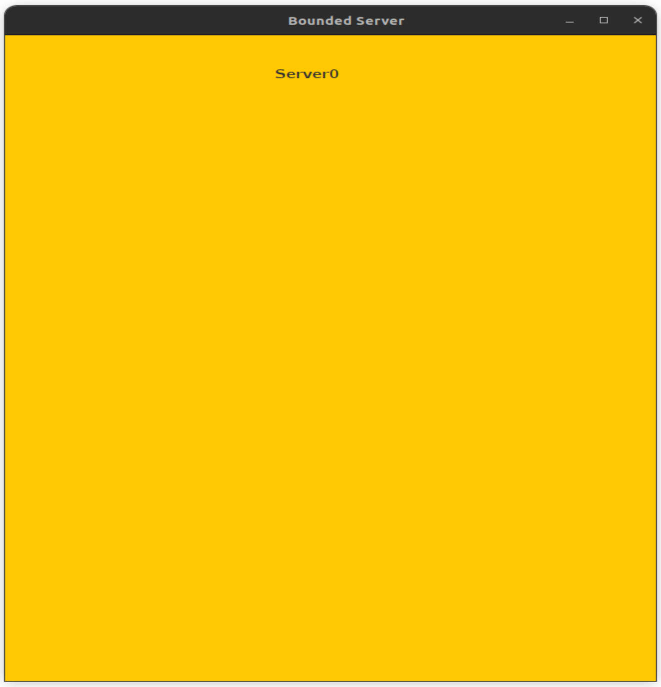


Figure 5.1 : Bounded BDLO once Launched. Only one server is shown.

Now we launch the Unbounded BDLO. This BDLO must also have 4 instances as presented in Figrue 5.2.

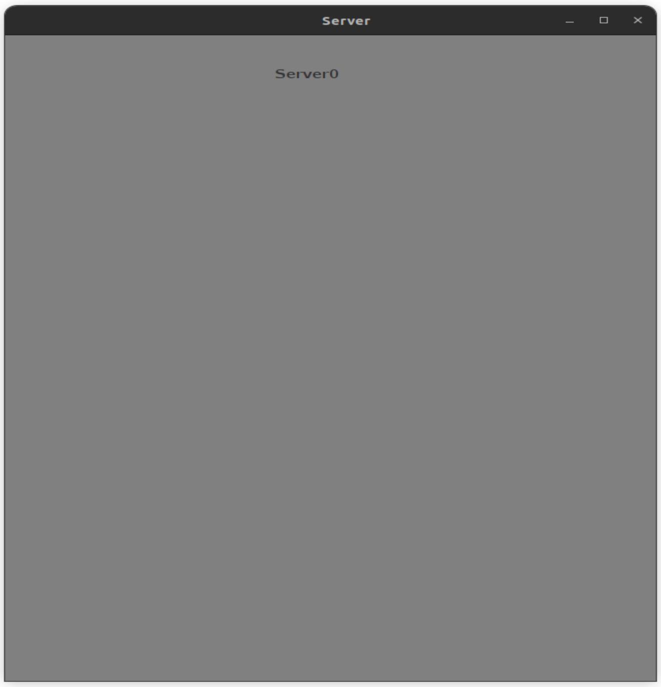
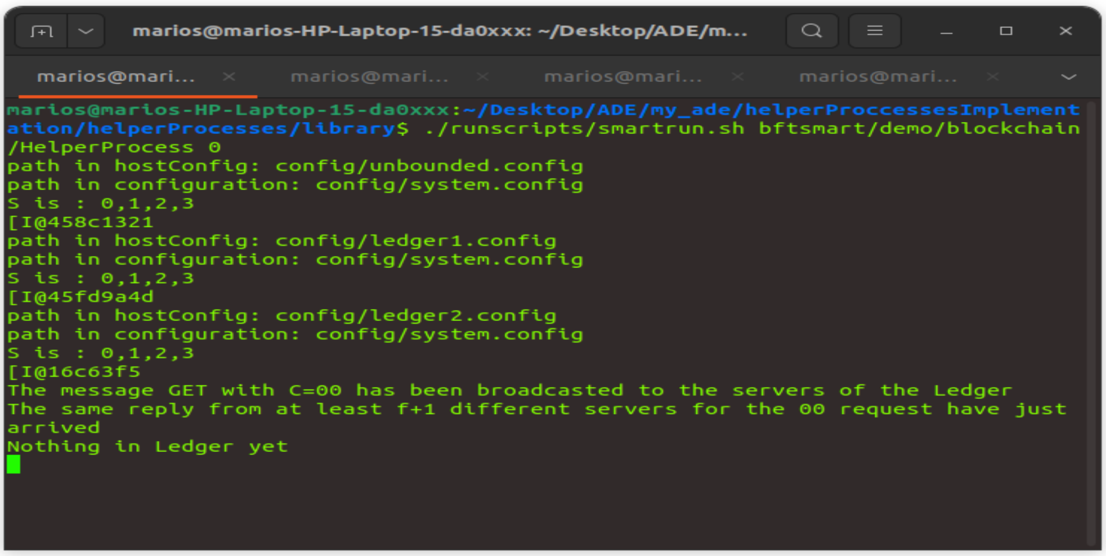


Figure 5.2 : Unbounded BDLO once Launched. Only one server is shown

After launching the BDLOs, we can start the helper processes. Figure 5.3 shows the terminal of the first helper process.



```
marios@marios-HP-Laptop-15-da0xxx:~/Desktop/ADE/my_ade/helperProccessesImplement
ation/helperProcesses/library$ ./runscripts/smartrun.sh bftsmart/demo/blockchain
/HelperProcess 0
path in hostConfig: config/unbounded.config
path in configuration: config/system.config
S is : 0,1,2,3
[I@458c1321
path in hostConfig: config/ledger1.config
path in configuration: config/system.config
S is : 0,1,2,3
[I@45fd9a4d
path in hostConfig: config/ledger2.config
path in configuration: config/system.config
S is : 0,1,2,3
[I@16c63f5
The message GET with C=00 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the 00 request have just
arrived
Nothing in Ledger yet
```

Figure 5.3 : Helper Processes once Launched. Only one Helper Process is shown.

Finally, we launch the clients. As seen in Figure 5.4 we launch the clients with ID 4.

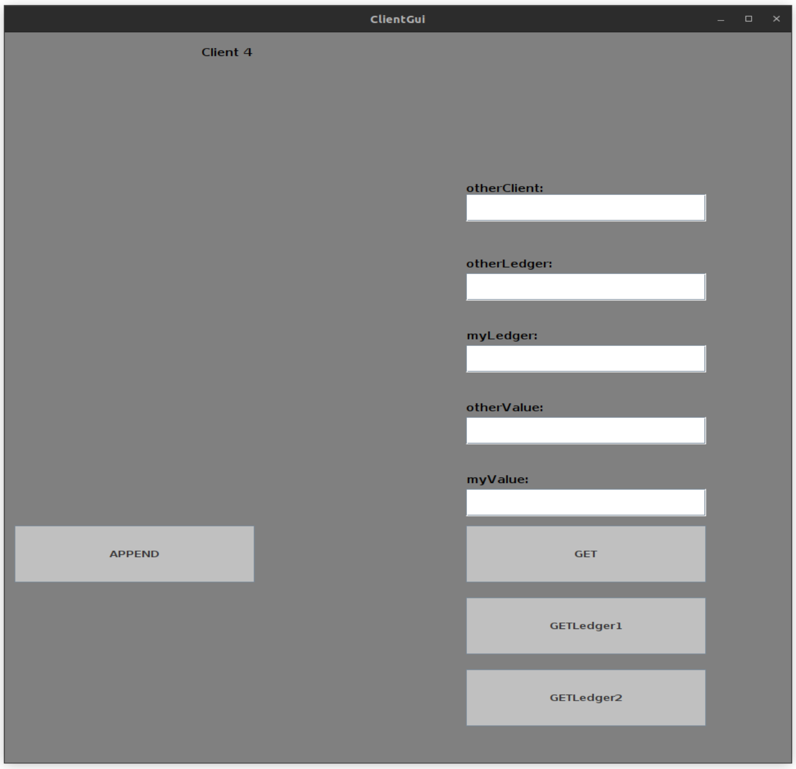


Figure 5.4 : Clients GUI of Client with ID 4. Client 5 is also launched but not shown

A client, in order to append a record, has to enter the following information :

- **otherClient** : The ID of the other client(In this scenario client 4 will write 5 and the opposite)
- **otherLedger :** In which ledger is the other client supposed to append
- **myLedger** : In which ledger the client is appending to
- **otherValue** : The expected value of the other's client record
- **myValue** : The clients value of the record

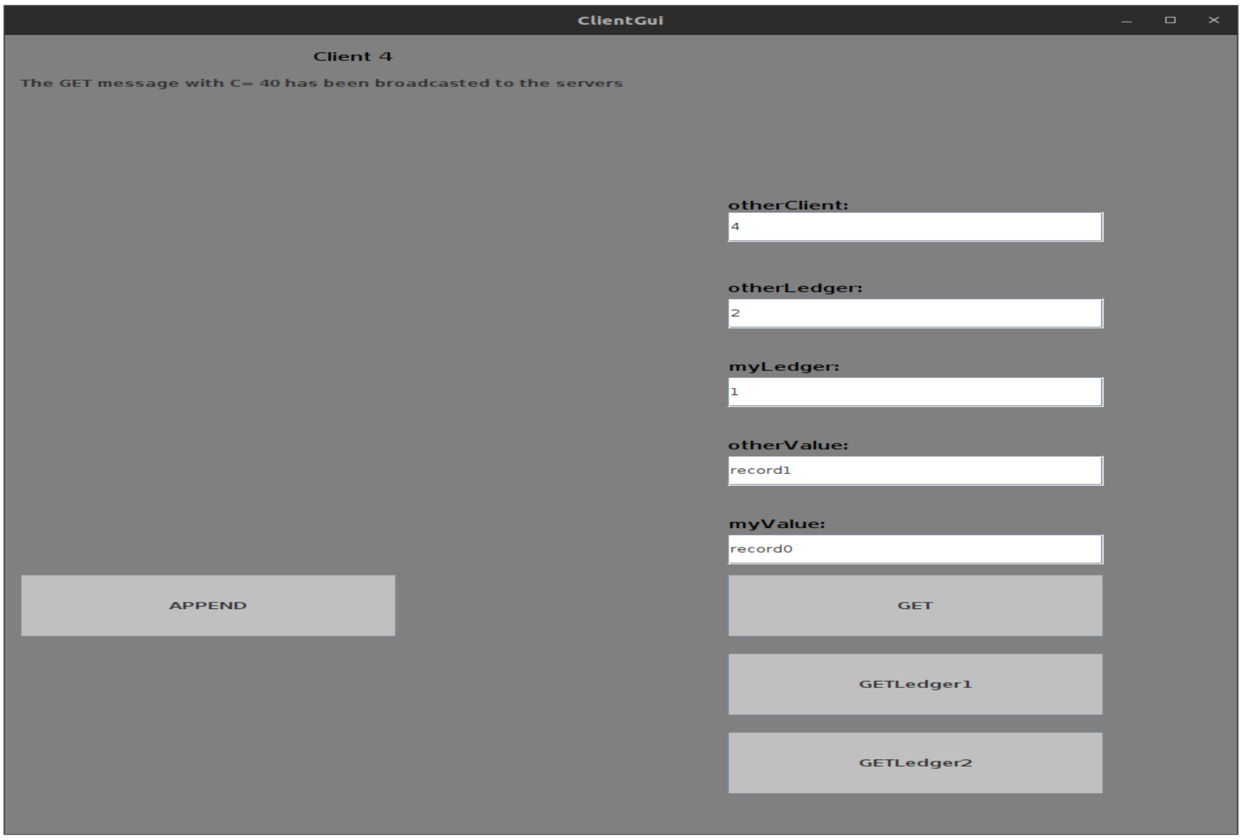
The GET button returns the Ledger of the unbounded BDLO, the GETLedger1 button returns the Ledger of the bounded BDLO with ID 1, and the GETLedger2 button returns the Ledger of the bounded BDLO with ID 2.

Now that everything is up and running, we can start appending Records.

Let's say the clients 4 and 5 have come to an agreement.

Client 4 wants to append a record with value "record0" to Ledger 1.

Client 5 wants to append a record with value "record1" to Ledger 2.



Client 4

The GET message with C= 40 has been broadcasted to the servers

otherClient: 4

otherLedger: 2

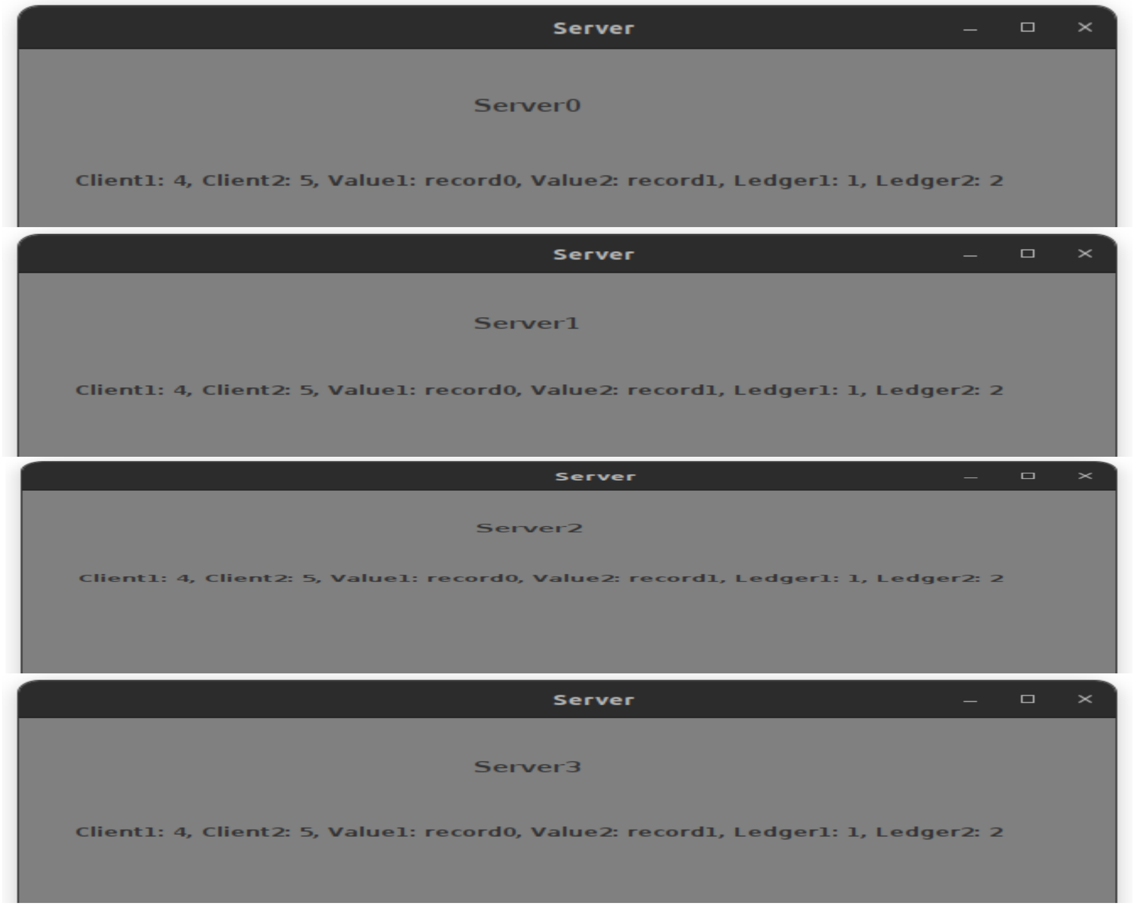
myLedger: 1

otherValue: record1

myValue: record0

Figure 5.5 shows the clients 4 append request structure.

Once the Append Request of Client 4 is complete, the unbounded BDLO must have one record, and the bounded BDLOs must have none. The following figures showcase this.



```
Server0

Client1: 4, Client2: 5, Value1: record0, Value2: record1, Ledger1: 1, Ledger2: 2
```

```
Server1

Client1: 4, Client2: 5, Value1: record0, Value2: record1, Ledger1: 1, Ledger2: 2
```

```
Server2

Client1: 4, Client2: 5, Value1: record0, Value2: record1, Ledger1: 1, Ledger2: 2
```

```
Server3

Client1: 4, Client2: 5, Value1: record0, Value2: record1, Ledger1: 1, Ledger2: 2
```

Figure 5.6 : Unbounded BDLO after Client's 4 Append Request

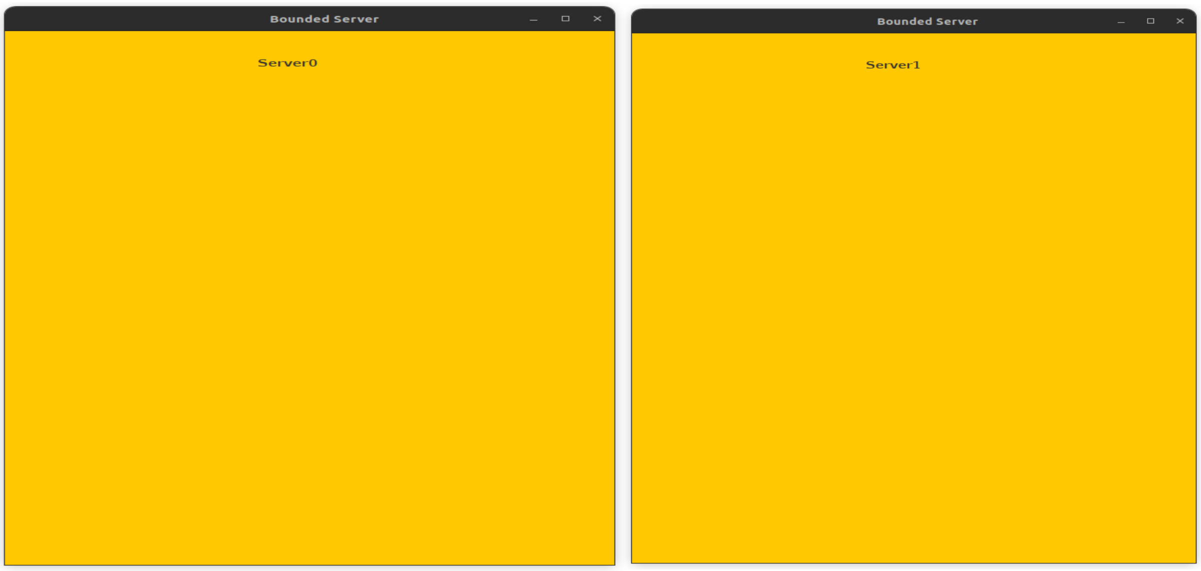


Figure 5.7 : Bounded BDLOs after Client 4 Append Request. Servers 0 and 1 (only one BDLO is shown)

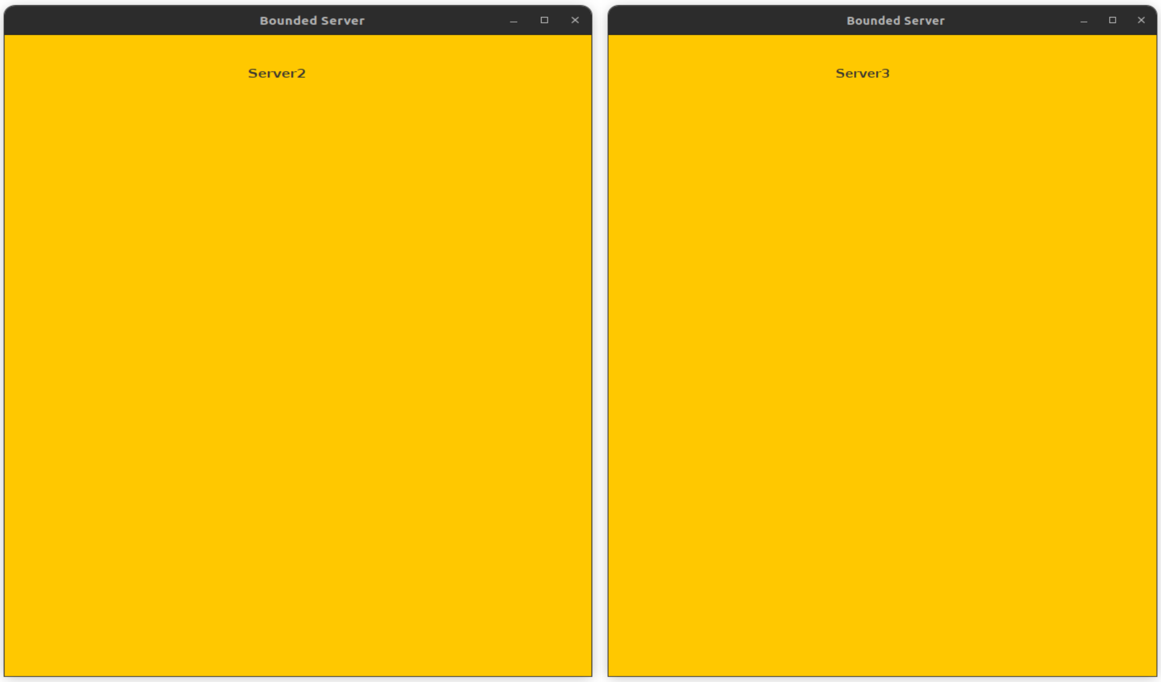


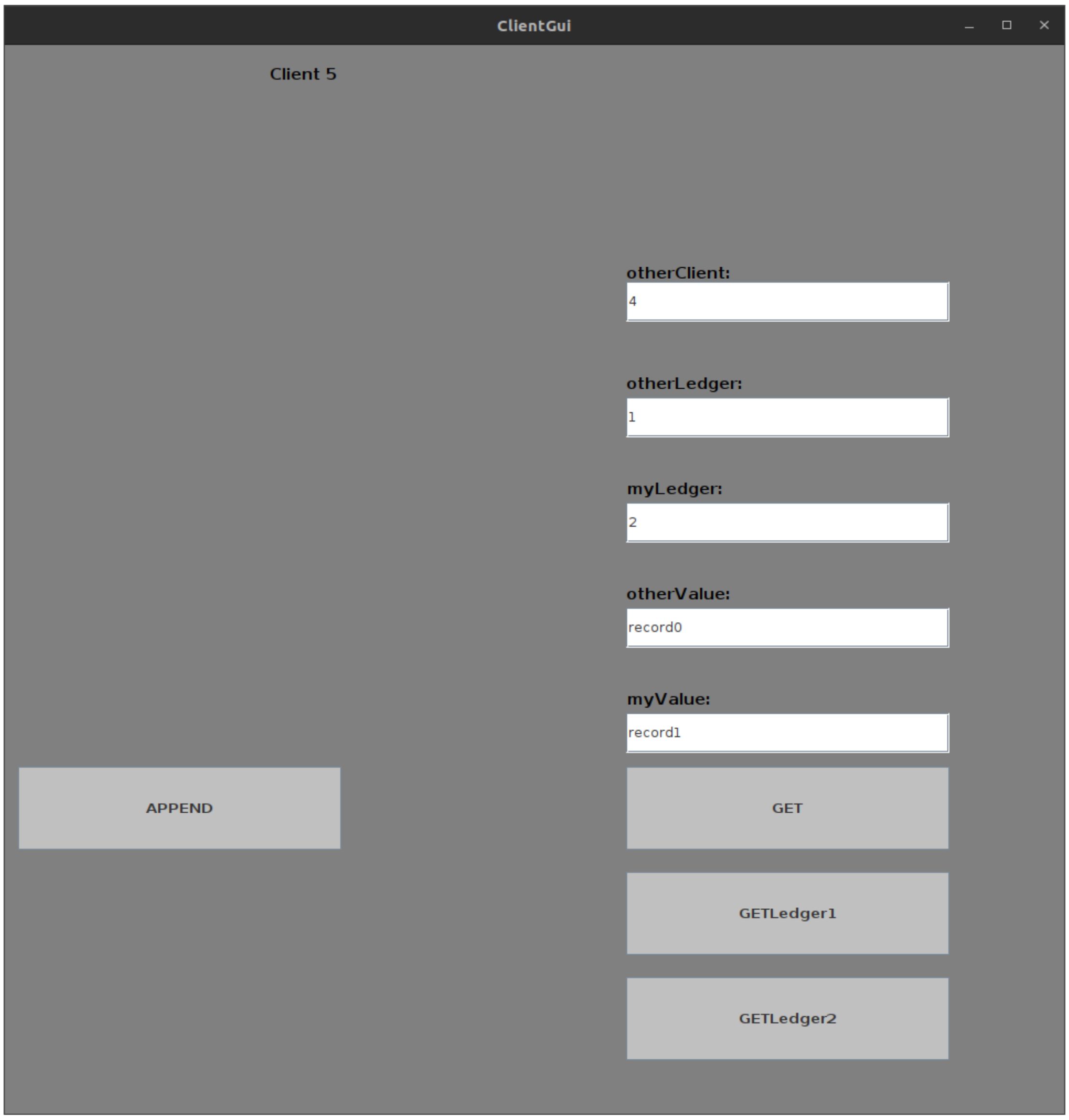
Figure 5.8 : Bounded BDLOs after Client 4 Append Request. Servers 2 and 3 (only one BDLO is shown)

In order for the exchange to be completed, client 5 must append their record as shown in Figure 5.9.

If even one of those attributes, do not correspond to what Client 4 has appended, then none of them is appended to the bounded BDLOs.

But since Client 5 appends what was agreed, then both Records are appended to their ledgers. The upcoming figures show how each record is appended.

Figure 5.10, shows the unbounded BDLO after we execute GET request from a client. As we can see both records are in the Ledger.



otherClient: 4

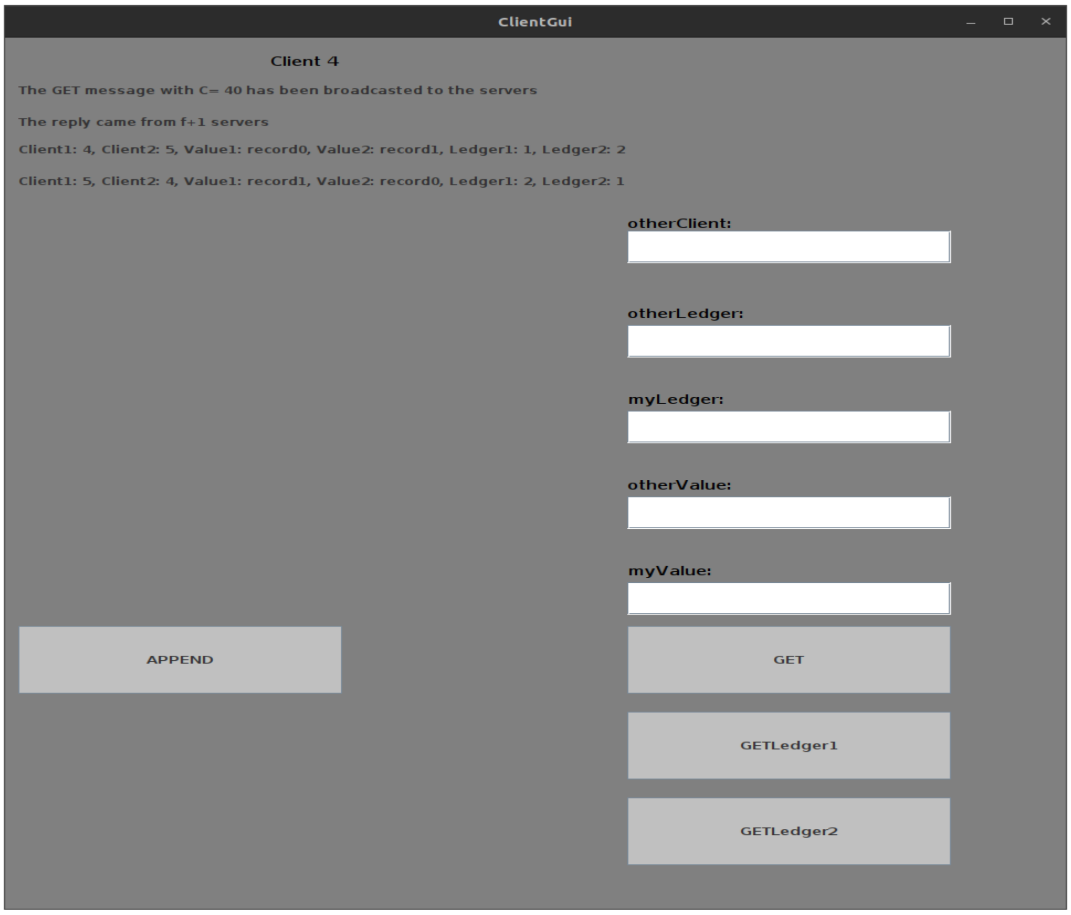
otherLedger: 1

myLedger: 2

otherValue: record0

myValue: record1

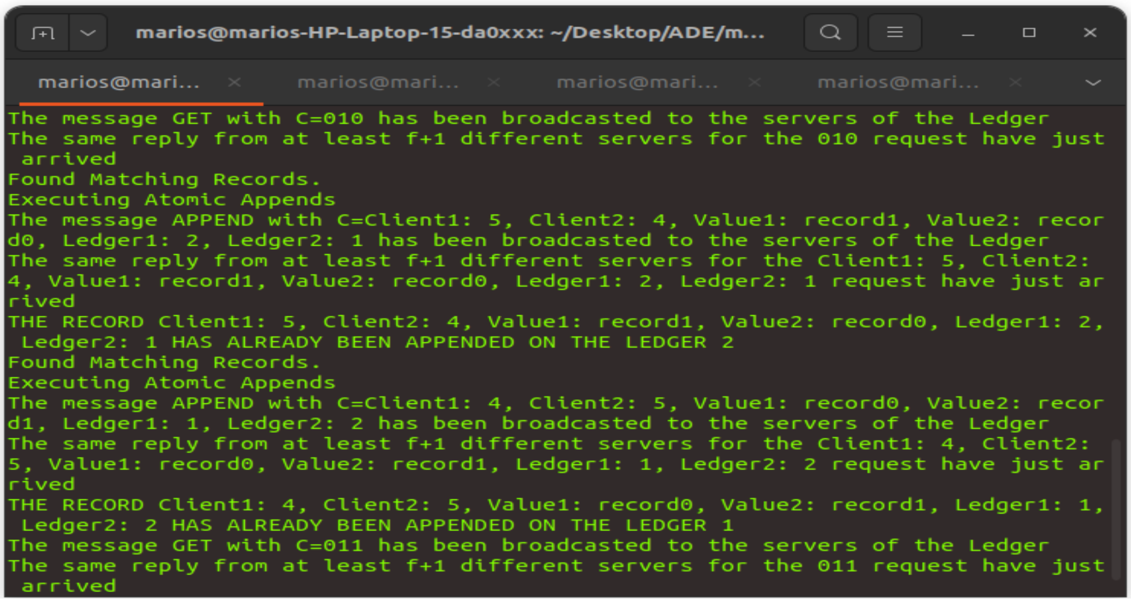
Figure 5.9 : The Append Request of Client 5



```
Client 4
The GET message with C= 40 has been broadcasted to the servers
The reply came from f+1 servers
Client1: 4, Client2: 5, Value1: record0, Value2: record1, Ledger1: 1, Ledger2: 2
Client1: 5, Client2: 4, Value1: record1, Value2: record0, Ledger1: 2, Ledger2: 1
```

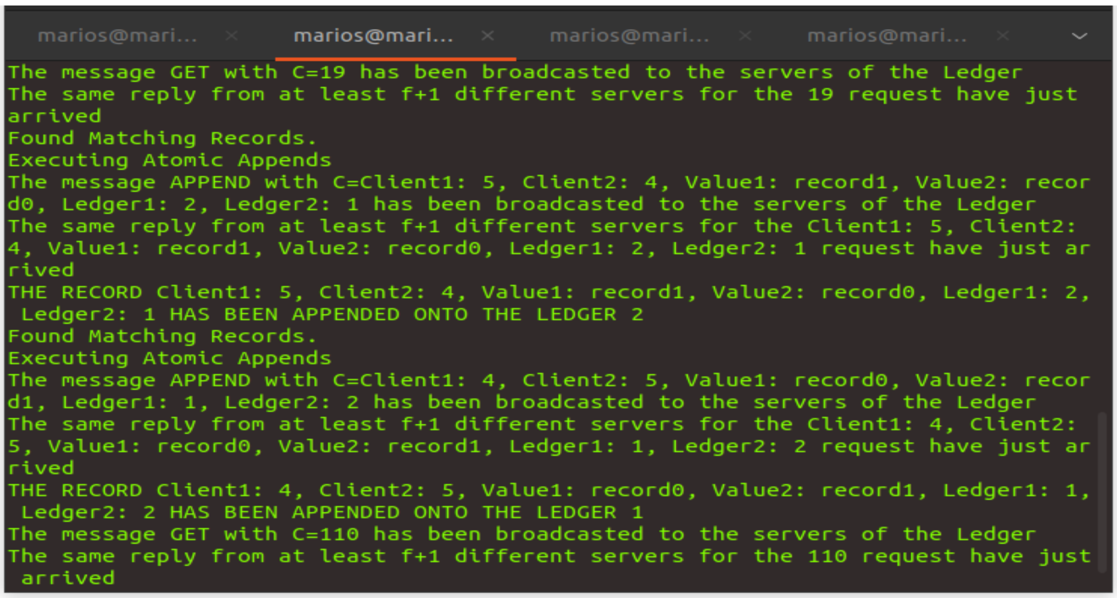
Figure 5.10 : GET request to the unbounded BDLO, once both Records have been Appended

Figures 5.11 and 5.12, show what happens when the helper processes discover the matching records, and execute the Append Requests towards the bounded BDLOs.

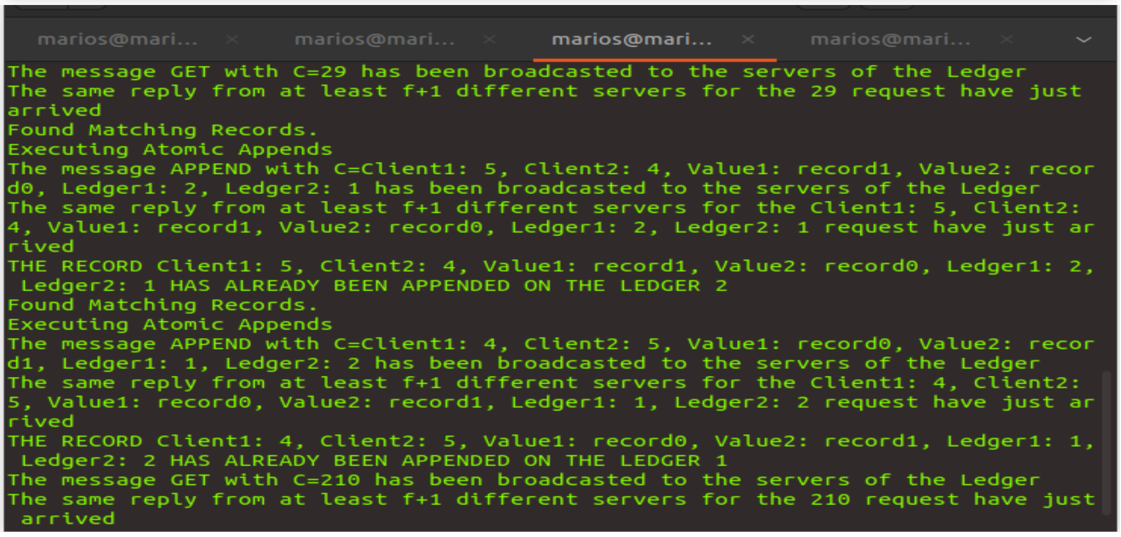


```
The message GET with C=010 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the 010 request have just
 arrived
Found Matching Records.
Executing Atomic Appends
The message APPEND with C=Client1: 5, Client2: 4, Value1: record1, Value2: recor
d0, Ledger1: 2, Ledger2: 1 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the Client1: 5, Client2:
4, Value1: record1, Value2: record0, Ledger1: 2, Ledger2: 1 request have just ar
rived
THE RECORD Client1: 5, Client2: 4, Value1: record1, Value2: record0, Ledger1: 2,
 Ledger2: 1 HAS ALREADY BEEN APPENDED ON THE LEDGER 2
Found Matching Records.
Executing Atomic Appends
The message APPEND with C=Client1: 4, Client2: 5, Value1: record0, Value2: recor
d1, Ledger1: 1, Ledger2: 2 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the Client1: 4, Client2:
5, Value1: record0, Value2: record1, Ledger1: 1, Ledger2: 2 request have just ar
rived
THE RECORD Client1: 4, Client2: 5, Value1: record0, Value2: record1, Ledger1: 1,
 Ledger2: 2 HAS ALREADY BEEN APPENDED ON THE LEDGER 1
The message GET with C=011 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the 011 request have just
 arrived
```

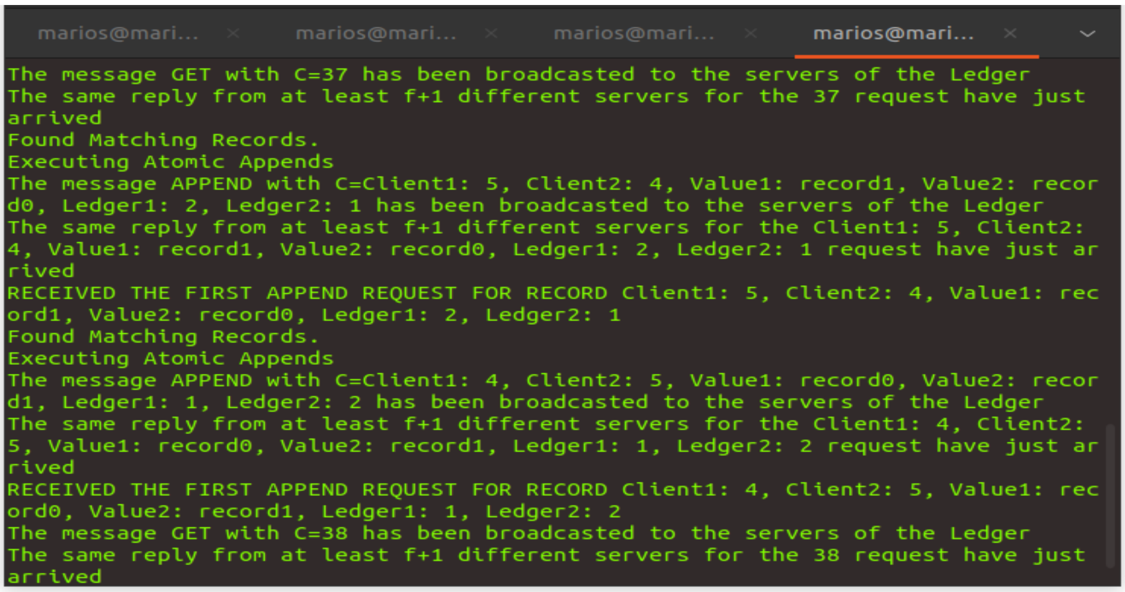
Figure 5.11 : Helper Process 1 Logs, once it discovers a Matching Pair



```
The message GET with C=19 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the 19 request have just
arrived
Found Matching Records.
Executing Atomic Appends
The message APPEND with C=Client1: 5, Client2: 4, Value1: record1, Value2: recor
d0, Ledger1: 2, Ledger2: 1 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the Client1: 5, Client2:
4, Value1: record1, Value2: record0, Ledger1: 2, Ledger2: 1 request have just ar
rived
THE RECORD Client1: 5, Client2: 4, Value1: record1, Value2: record0, Ledger1: 2,
 Ledger2: 1 HAS BEEN APPENDED ONTO THE LEDGER 2
Found Matching Records.
Executing Atomic Appends
The message APPEND with C=Client1: 4, Client2: 5, Value1: record0, Value2: recor
d1, Ledger1: 1, Ledger2: 2 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the Client1: 4, Client2:
5, Value1: record0, Value2: record1, Ledger1: 1, Ledger2: 2 request have just ar
rived
THE RECORD Client1: 4, Client2: 5, Value1: record0, Value2: record1, Ledger1: 1,
 Ledger2: 2 HAS BEEN APPENDED ONTO THE LEDGER 1
The message GET with C=110 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the 110 request have just
 arrived
```



```
The message GET with C=29 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the 29 request have just
arrived
Found Matching Records.
Executing Atomic Appends
The message APPEND with C=Client1: 5, Client2: 4, Value1: record1, Value2: recor
d0, Ledger1: 2, Ledger2: 1 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the Client1: 5, Client2:
4, Value1: record1, Value2: record0, Ledger1: 2, Ledger2: 1 request have just ar
rived
THE RECORD Client1: 5, Client2: 4, Value1: record1, Value2: record0, Ledger1: 2,
 Ledger2: 1 HAS ALREADY BEEN APPENDED ON THE LEDGER 2
Found Matching Records.
Executing Atomic Appends
The message APPEND with C=Client1: 4, Client2: 5, Value1: record0, Value2: recor
d1, Ledger1: 1, Ledger2: 2 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the Client1: 4, Client2:
5, Value1: record0, Value2: record1, Ledger1: 1, Ledger2: 2 request have just ar
rived
THE RECORD Client1: 4, Client2: 5, Value1: record0, Value2: record1, Ledger1: 1,
 Ledger2: 2 HAS ALREADY BEEN APPENDED ON THE LEDGER 1
The message GET with C=210 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the 210 request have just
 arrived
```

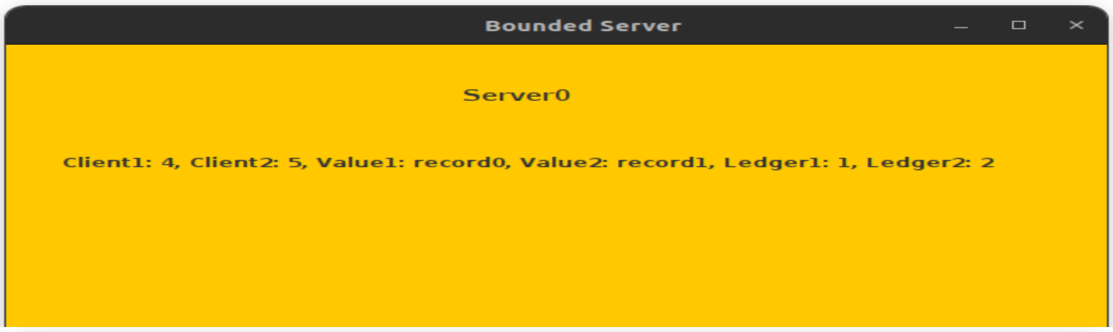


```
The message GET with C=37 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the 37 request have just
arrived
Found Matching Records.
Executing Atomic Appends
The message APPEND with C=Client1: 5, Client2: 4, Value1: record1, Value2: recor
d0, Ledger1: 2, Ledger2: 1 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the Client1: 5, Client2:
4, Value1: record1, Value2: record0, Ledger1: 2, Ledger2: 1 request have just ar
rived
RECEIVED THE FIRST APPEND REQUEST FOR RECORD Client1: 5, Client2: 4, Value1: rec
ord1, Value2: record0, Ledger1: 2, Ledger2: 1
Found Matching Records.
Executing Atomic Appends
The message APPEND with C=Client1: 4, Client2: 5, Value1: record0, Value2: recor
d1, Ledger1: 1, Ledger2: 2 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the Client1: 4, Client2:
5, Value1: record0, Value2: record1, Ledger1: 1, Ledger2: 2 request have just ar
rived
RECEIVED THE FIRST APPEND REQUEST FOR RECORD Client1: 4, Client2: 5, Value1: rec
ord0, Value2: record1, Ledger1: 1, Ledger2: 2
The message GET with C=38 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the 38 request have just
arrived
```

Figure 5.12 : Helper Processes 2,3 and 4 Logs, once they discover a Matching Pair

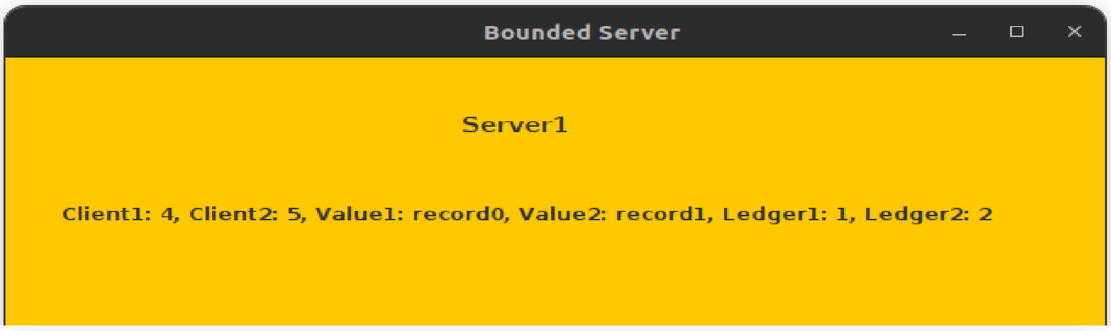
As we can see in Figures 5.11 and 5.12, there are 4 helper processes. So only one of them can be byzantine, and the records are appended on the bounded BDLOs, once they receive two Append Requests(t+1 requests) for the same record. The 4th Helper Process, was the first to send the Append Requests, hence the message that we see "RECEIVED THE FIRST APPEND REQUEST FOR RECORD…". The 2nd Helper Process was the one to send the second request, and that's why we see the messages "THE RECORD … HAS BEEN APPENDED ONTO THE LEDGER X". The 1st and 3rd Helper Processes sent their request after the records were appended, so they get the message "THE RECORD … HAS ALREADY BEEN APPENDED ONTO THE LEDGER X".

Finally, we can see the bounded BDLOs. Figure 5.13 and 5.14 show the BDLO with ID 1, and Figures 5.15 and 5.16 show the BDLO with ID 2.



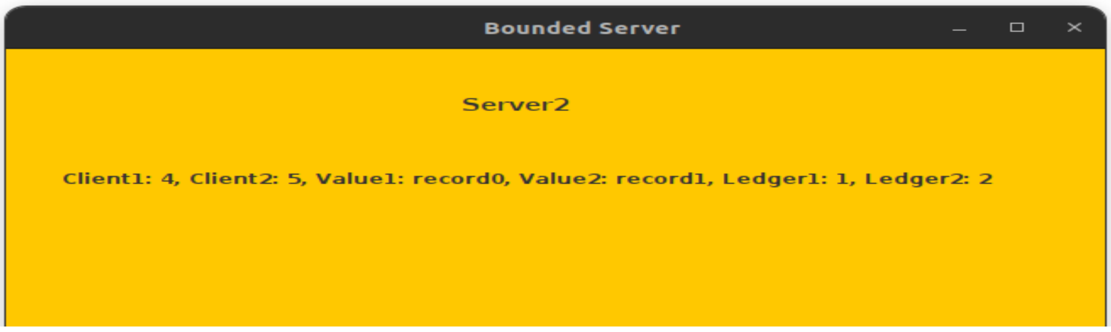
Server0

```
Client1: 4, Client2: 5, Value1: record0, Value2: record1, Ledger1: 1, Ledger2: 2
```



Server1

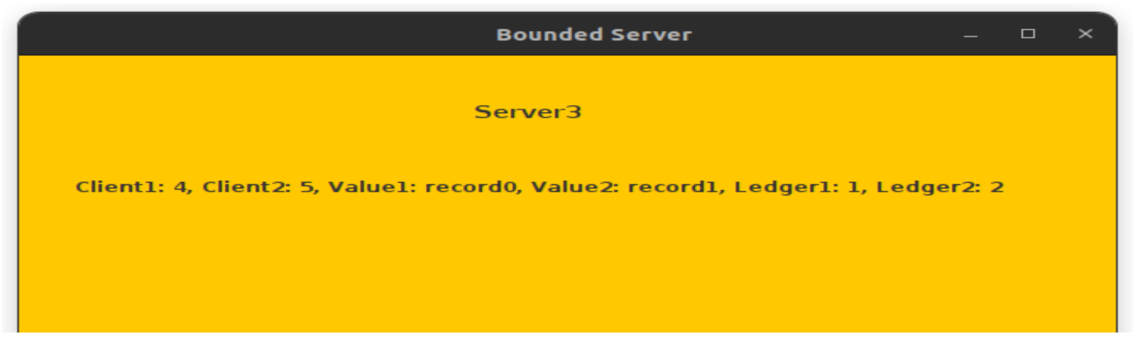
```
Client1: 4, Client2: 5, Value1: record0, Value2: record1, Ledger1: 1, Ledger2: 2
```



Server2

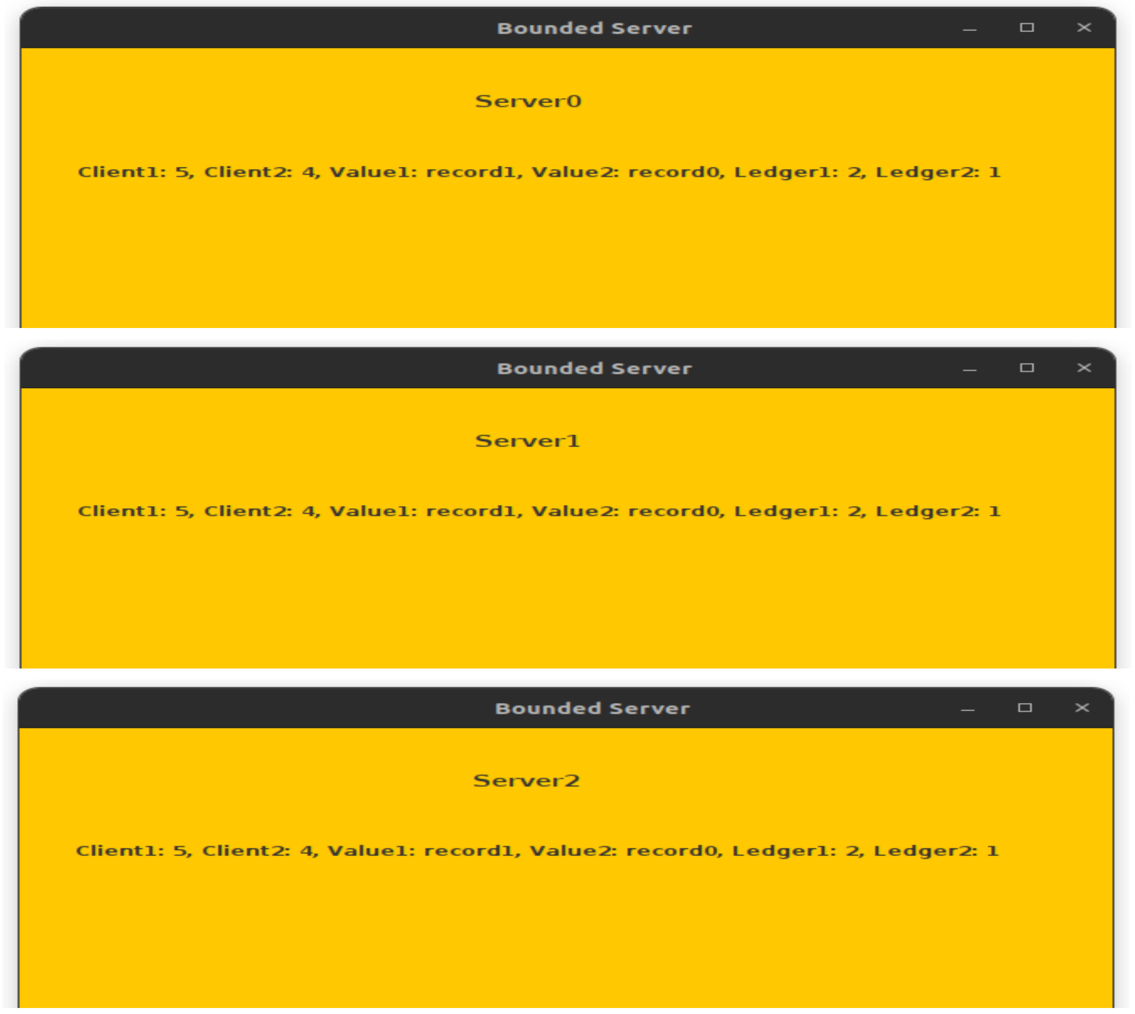
```
Client1: 4, Client2: 5, Value1: record0, Value2: record1, Ledger1: 1, Ledger2: 2
```

Figure 5.13 : The bounded-BDLO with ID 1, after the Atomic Appends execution from the Helper Processes (only servers 0,1 and 2 are shown)



```
Client1: 4, Client2: 5, Value1: record0, Value2: record1, Ledger1: 1, Ledger2: 2
```

Figure 5.14 : The bounded-BDLO with ID 1, after the Atomic Appends execution from the Helper Processes (only server 3 is shown)

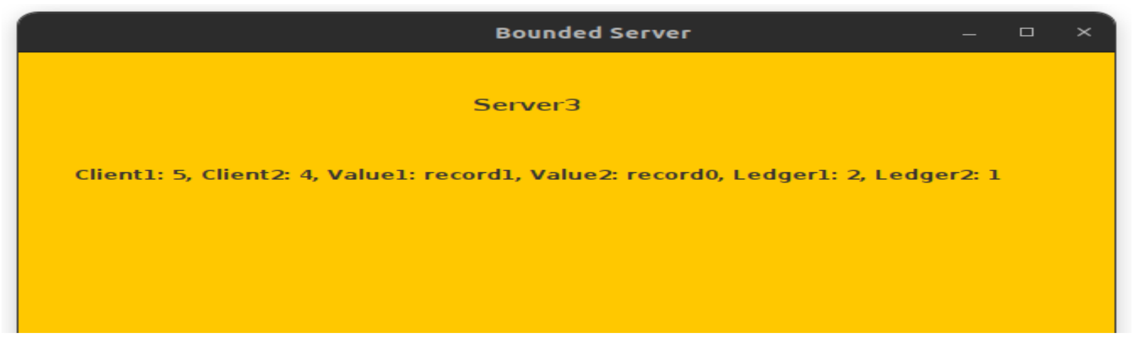


```
Client1: 5, Client2: 4, Value1: record1, Value2: record0, Ledger1: 2, Ledger2: 1
```

```
Client1: 5, Client2: 4, Value1: record1, Value2: record0, Ledger1: 2, Ledger2: 1
```

```
Client1: 5, Client2: 4, Value1: record1, Value2: record0, Ledger1: 2, Ledger2: 1
```

Figure 5.15 : The bounded-BDLO with ID 2, after the Atomic Appends execution from the Helper Processes (only servers 0,1 and 2 are shown)



```
Client1: 5, Client2: 4, Value1: record1, Value2: record0, Ledger1: 2, Ledger2: 1
```

Figure 5.16 : The bounded-BDLO with ID 2, after the Atomic Appends execution from the Helper Processes (only server 3 is shown)

5.3 Smart BDLO Execution

After demonstrating the Helper Processes implementation, we can can carry on with the Smart-BDLO implementation.

5.3.1 Instructions on How to Launch the System

The instructions are very similar to those in Section 5.2.1. Again in this scenario, the bounded BDLOs consist of four servers each. The Smart-BDLO – which replaces the Unbounded BDLO – also consists of four serves.

We must configure the bounded BDLOs and the clients, the same way they were configured in chapter 5.1. As for the Smart BDLO, we need to go the directory sbdlo/smart-BDLO/library/config and configure system.config, hosts.config(the servers of the smbdlo), ledger1.config and ledger2.config.

Order to launch each instance:

1. bounded-BDLOs
2. Smart-BDLO
3. Clients

The bounded-BDLOs and Clients can be started the same way as we saw before(they can also be found in the sbdlo folder). In order to launch the Smart-BDLO, we need to go the directory sbdlo/smart-BDLO/library and execute the same commands we did for the unbounded BDLO.

5.3.2 Demonstration of Smart BDLO Implementation

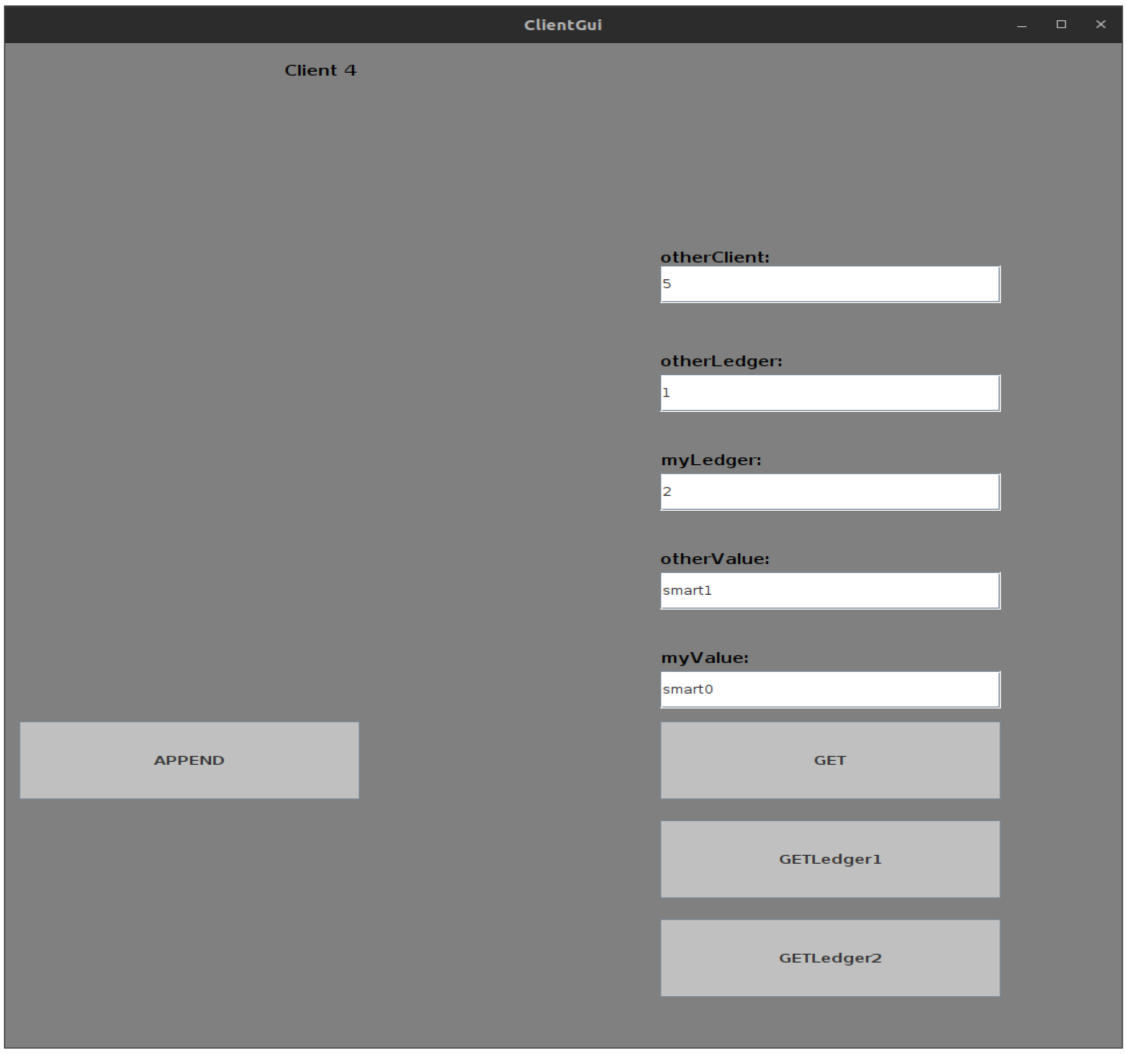
All the interfaces look the same as they did in Helper Processes implementation. As for the Smart-BDLO, it has the same interface as the unbouded-BDLO.

Let's say the clients 4 and 5 have come to an agreement.

Client 4 wants to append a record with value "smart0" to the Ledger 2.

Client 5 wants to append a record with value "smart1" to the Ledger 1.

Their Append Requests can be seen in Figure 5.14 and 5.18 respectively.



ClientGui

Client 4

otherClient: 5

otherLedger: 1

myLedger: 2

otherValue: smart1

myValue: smart0

Figure 5.17 : Client's 4 Append Request

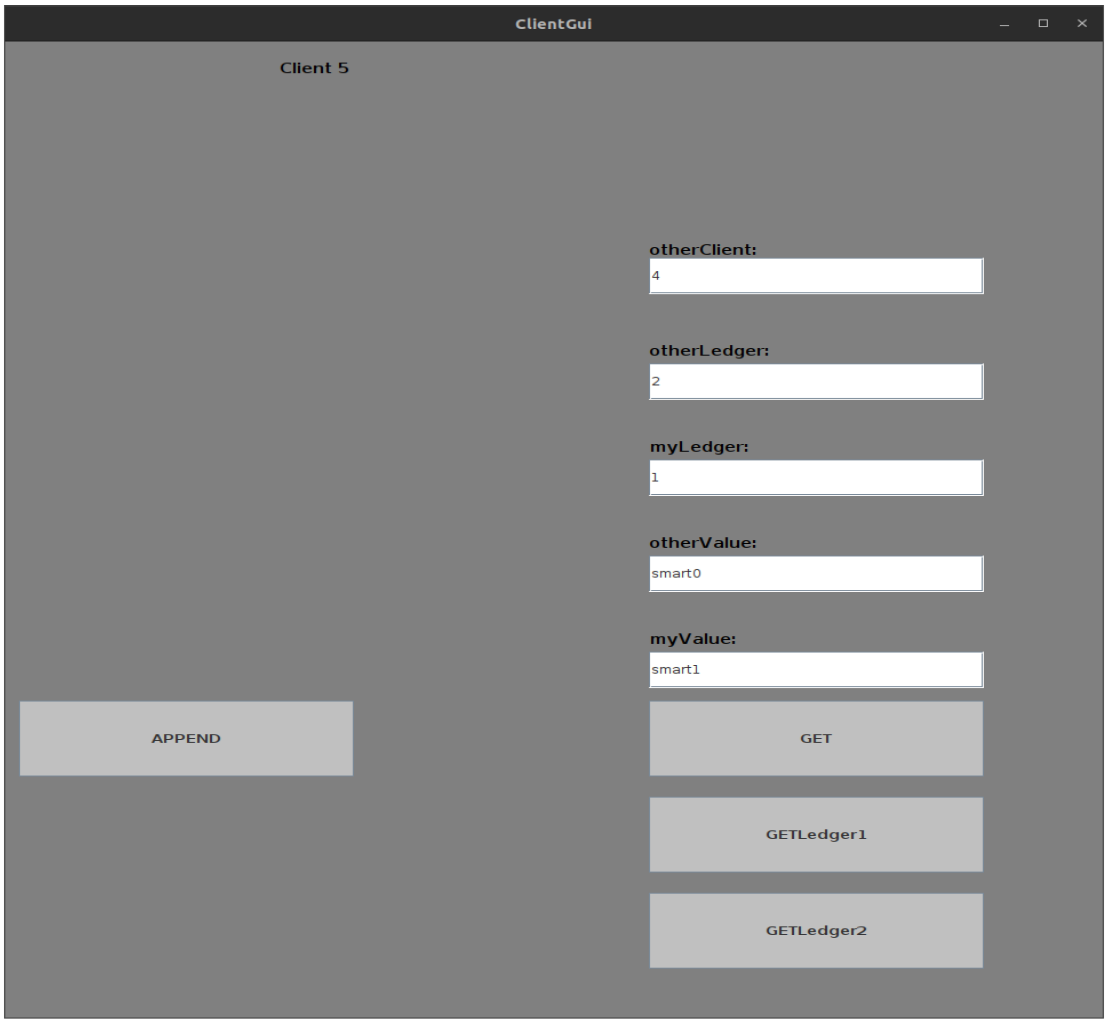
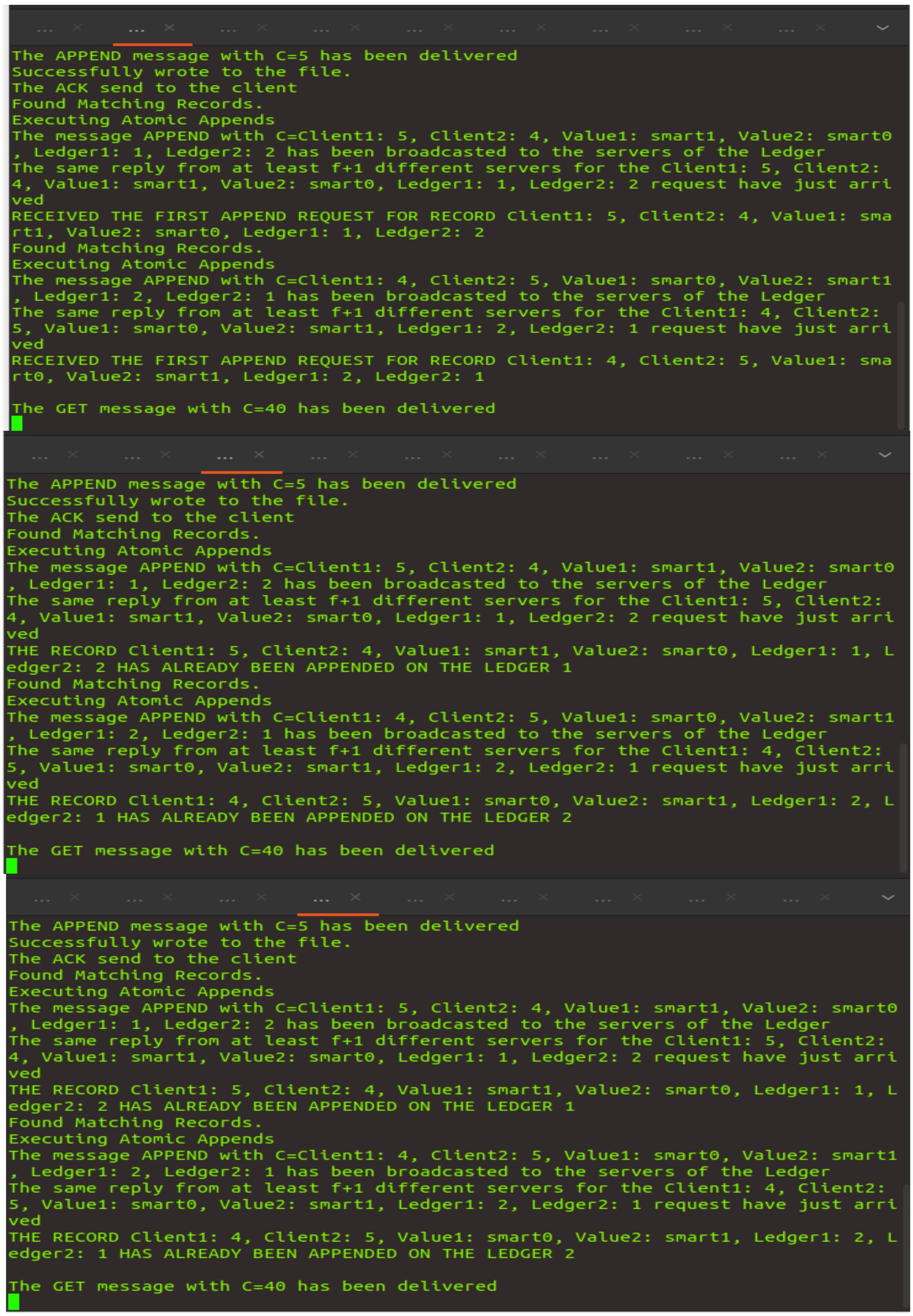


Figure 5.18 : Client's 5 Append Request

Once the smart BDLO detects the matching record, it starts executing the Append Requests. Figures 5.19 and 5.20, show the logs of the servers that the Smart BDLO consists of, once they detect the matching records. As it is presented, the 1st server of the Smart BDLO, was the first to execute the Append Requests. Hence the message it receives "RECEIVED THE FIRST APPEND REQUEST FOR RECORD …". The 4th server was the second one to send it's Append Request, and it was this request that made the bounded BDLOs to append the records on their Ledger. Consequently the server received the message "THE RECORD … HAS BEEN APPENDED ON THE LEDGER X". The 2nd and 3rd server sent their requests after the records were appended, so they received the message "THE RECORD … HAS ALREADY BEEN APPENDED ON THE LEDGER X".



```
The APPEND message with C=5 has been delivered
Successfully wrote to the file.
The ACK send to the client
Found Matching Records.
Executing Atomic Appends
The message APPEND with C=Client1: 5, Client2: 4, Value1: smart1, Value2: smart0
, Ledger1: 1, Ledger2: 2 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the Client1: 5, Client2:
4, Value1: smart1, Value2: smart0, Ledger1: 1, Ledger2: 2 request have just arri
ved
RECEIVED THE FIRST APPEND REQUEST FOR RECORD Client1: 5, Client2: 4, Value1: sma
rt1, Value2: smart0, Ledger1: 1, Ledger2: 2
Found Matching Records.
Executing Atomic Appends
The message APPEND with C=Client1: 4, Client2: 5, Value1: smart0, Value2: smart1
, Ledger1: 2, Ledger2: 1 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the Client1: 4, Client2:
5, Value1: smart0, Value2: smart1, Ledger1: 2, Ledger2: 1 request have just arri
ved
RECEIVED THE FIRST APPEND REQUEST FOR RECORD Client1: 4, Client2: 5, Value1: sma
rt0, Value2: smart1, Ledger1: 2, Ledger2: 1

The GET message with C=40 has been delivered
```

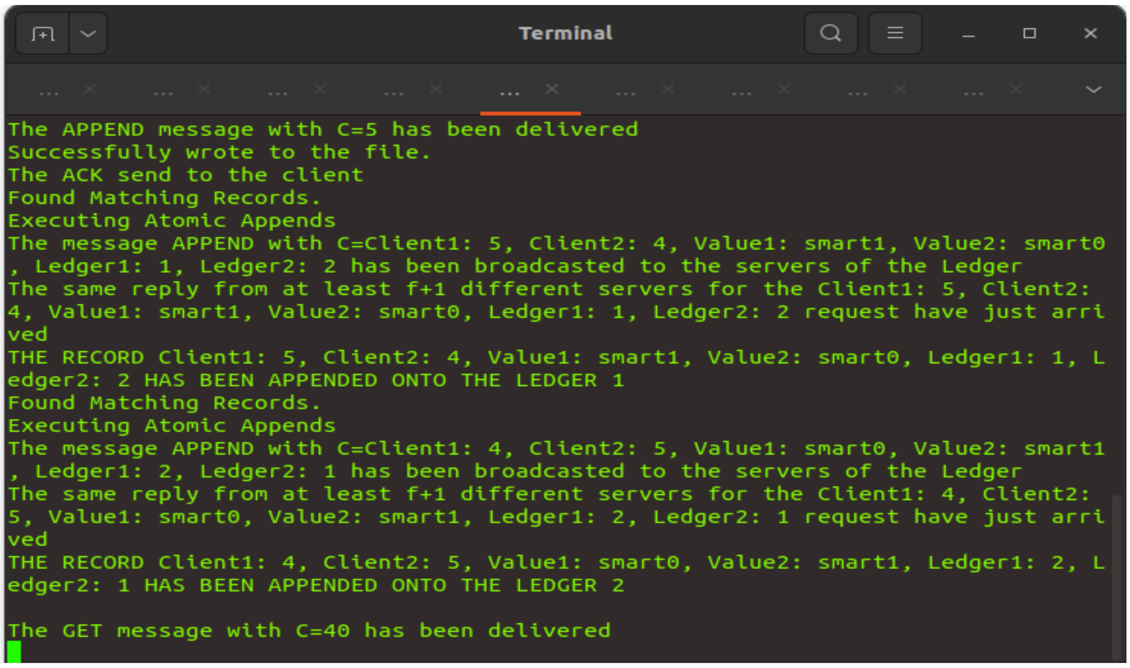
```
The APPEND message with C=5 has been delivered
Successfully wrote to the file.
The ACK send to the client
Found Matching Records.
Executing Atomic Appends
The message APPEND with C=Client1: 5, Client2: 4, Value1: smart1, Value2: smart0
, Ledger1: 1, Ledger2: 2 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the Client1: 5, Client2:
4, Value1: smart1, Value2: smart0, Ledger1: 1, Ledger2: 2 request have just arri
ved
THE RECORD Client1: 5, Client2: 4, Value1: smart1, Value2: smart0, Ledger1: 1, L
edger2: 2 HAS ALREADY BEEN APPENDED ON THE LEDGER 1
Found Matching Records.
Executing Atomic Appends
The message APPEND with C=Client1: 4, Client2: 5, Value1: smart0, Value2: smart1
, Ledger1: 2, Ledger2: 1 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the Client1: 4, Client2:
5, Value1: smart0, Value2: smart1, Ledger1: 2, Ledger2: 1 request have just arri
ved
THE RECORD Client1: 4, Client2: 5, Value1: smart0, Value2: smart1, Ledger1: 2, L
edger2: 1 HAS ALREADY BEEN APPENDED ON THE LEDGER 2

The GET message with C=40 has been delivered
```

```
The APPEND message with C=5 has been delivered
Successfully wrote to the file.
The ACK send to the client
Found Matching Records.
Executing Atomic Appends
The message APPEND with C=Client1: 5, Client2: 4, Value1: smart1, Value2: smart0
, Ledger1: 1, Ledger2: 2 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the Client1: 5, Client2:
4, Value1: smart1, Value2: smart0, Ledger1: 1, Ledger2: 2 request have just arri
ved
THE RECORD Client1: 5, Client2: 4, Value1: smart1, Value2: smart0, Ledger1: 1, L
edger2: 2 HAS ALREADY BEEN APPENDED ON THE LEDGER 1
Found Matching Records.
Executing Atomic Appends
The message APPEND with C=Client1: 4, Client2: 5, Value1: smart0, Value2: smart1
, Ledger1: 2, Ledger2: 1 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the Client1: 4, Client2:
5, Value1: smart0, Value2: smart1, Ledger1: 2, Ledger2: 1 request have just arri
ved
THE RECORD Client1: 4, Client2: 5, Value1: smart0, Value2: smart1, Ledger1: 2, L
edger2: 1 HAS ALREADY BEEN APPENDED ON THE LEDGER 2

The GET message with C=40 has been delivered
```

Figure 5.19 : Smart BDLO Logs once it identifies the Matching Records (only servers 0,1 and 2 are shown)

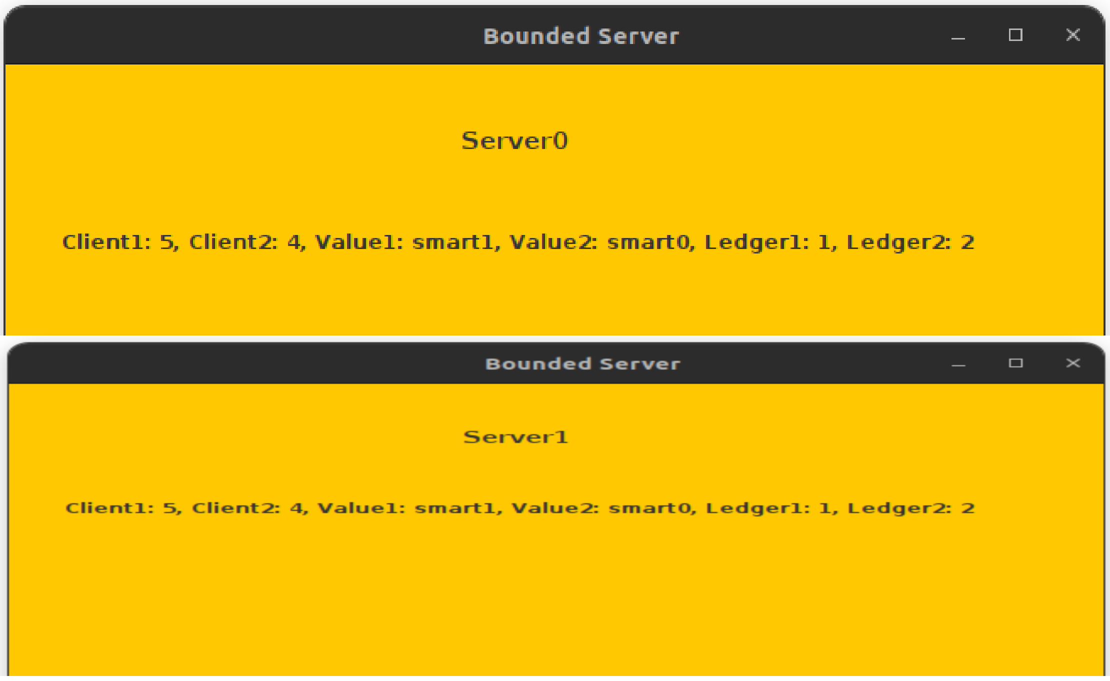


```
The APPEND message with C=5 has been delivered
Successfully wrote to the file.
The ACK send to the client
Found Matching Records.
Executing Atomic Appends
The message APPEND with C=Client1: 5, Client2: 4, Value1: smart1, Value2: smart0
, Ledger1: 1, Ledger2: 2 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the Client1: 5, Client2:
4, Value1: smart1, Value2: smart0, Ledger1: 1, Ledger2: 2 request have just arri
ved
THE RECORD Client1: 5, Client2: 4, Value1: smart1, Value2: smart0, Ledger1: 1, L
edger2: 2 HAS BEEN APPENDED ONTO THE LEDGER 1
Found Matching Records.
Executing Atomic Appends
The message APPEND with C=Client1: 4, Client2: 5, Value1: smart0, Value2: smart1
, Ledger1: 2, Ledger2: 1 has been broadcasted to the servers of the Ledger
The same reply from at least f+1 different servers for the Client1: 4, Client2:
5, Value1: smart0, Value2: smart1, Ledger1: 2, Ledger2: 1 request have just arri
ved
THE RECORD Client1: 4, Client2: 5, Value1: smart0, Value2: smart1, Ledger1: 2, L
edger2: 1 HAS BEEN APPENDED ONTO THE LEDGER 2

The GET message with C=40 has been delivered
```

Figure 5.20 : Smart BDLO Logs once it identifies the Matching Records (only server 3 is shown)

Finally we can see in the upcoming figures, that each record has been appended on the correct bounded BDLO.



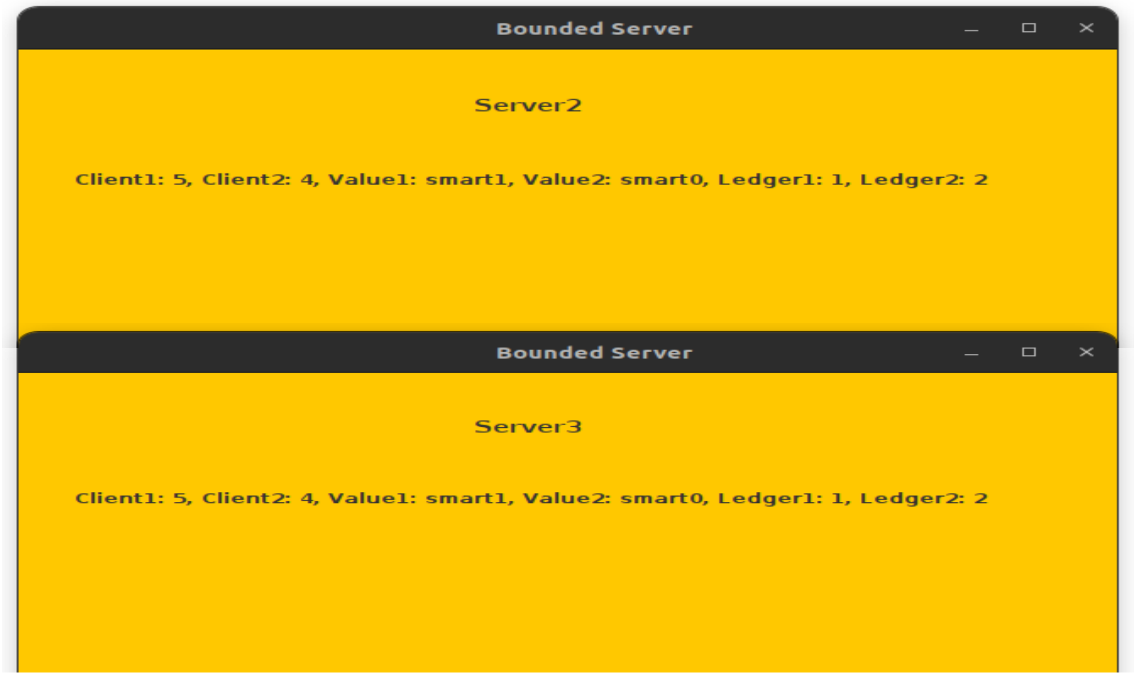
Server0

Client1: 5, Client2: 4, Value1: smart1, Value2: smart0, Ledger1: 1, Ledger2: 2

Server1

Client1: 5, Client2: 4, Value1: smart1, Value2: smart0, Ledger1: 1, Ledger2: 2

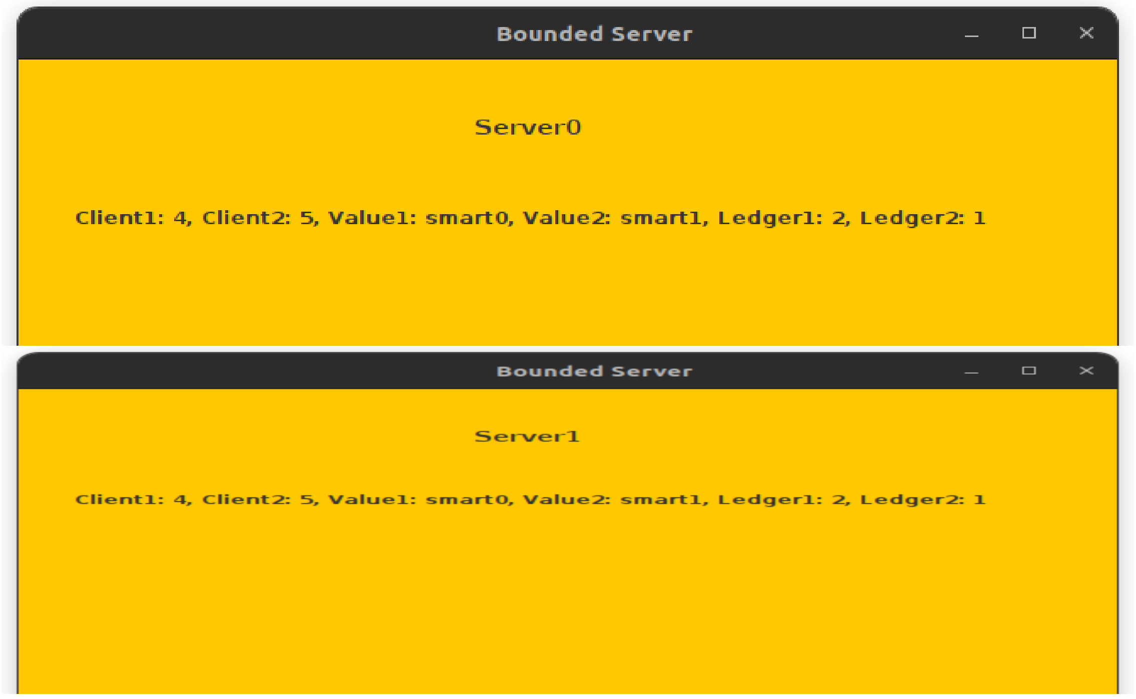
Figure 5.21 : The bounded-BDLO with ID 1, after the Atomic Appends execution from the Smart BDLO (only servers 0 and 1 are shown)



```
Client1: 5, Client2: 4, Value1: smart1, Value2: smart0, Ledger1: 1, Ledger2: 2
```

```
Client1: 5, Client2: 4, Value1: smart1, Value2: smart0, Ledger1: 1, Ledger2: 2
```

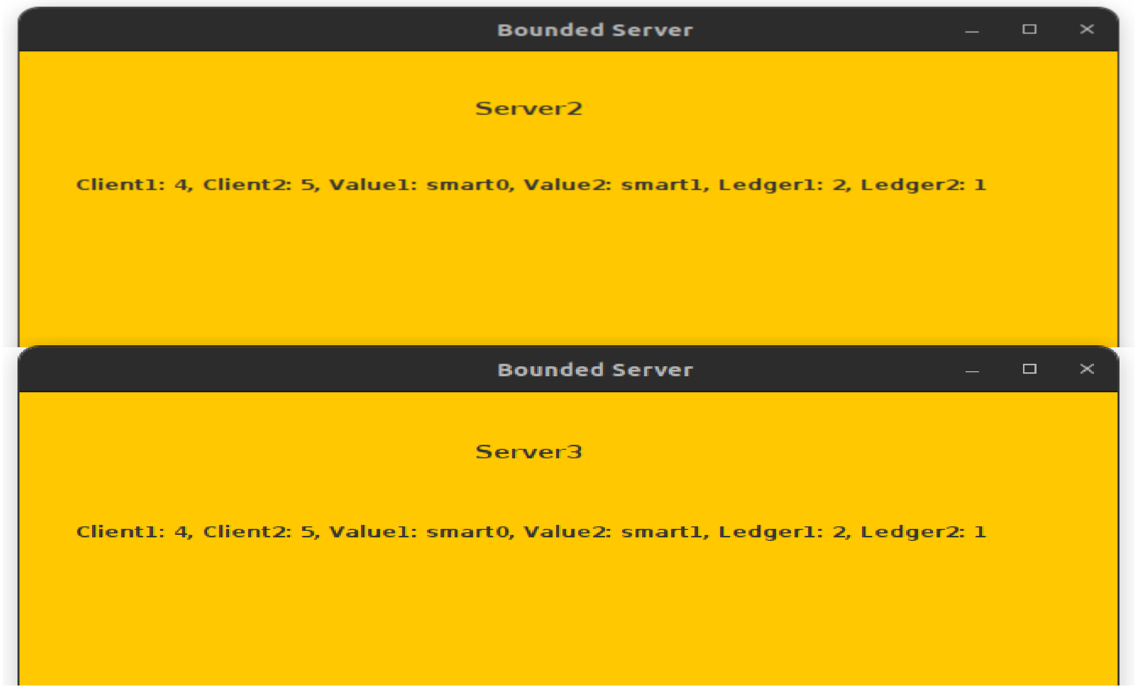
Figure 5.22 : The bounded-BDLO with ID 1, after the Atomic Appends execution from the Smart BDLO (only servers 2 and 3 are shown)



```
Client1: 4, Client2: 5, Value1: smart0, Value2: smart1, Ledger1: 2, Ledger2: 1
```

```
Client1: 4, Client2: 5, Value1: smart0, Value2: smart1, Ledger1: 2, Ledger2: 1
```

Figure 5.23 : The bounded-BDLO with ID 2, after the Atomic Appends execution from the Smart BDLO (only servers 0 and 1 are shown)



Server2

```
Client1: 4, Client2: 5, Value1: smart0, Value2: smart1, Ledger1: 2, Ledger2: 1
```

Server3

```
Client1: 4, Client2: 5, Value1: smart0, Value2: smart1, Ledger1: 2, Ledger2: 1
```

Figure 5.24 : The bounded-BDLO with ID 2, after the Atomic Appends execution from the Smart BDLO (only servers 2 and 3 are shown)

5.4 Comparing the Performance of the Algorithms

The general idea for both algorithms is to use a distributed middleman that searches the records the clients append, to find the matching ones. The algorithms are pretty similar with each other, with the main difference being that a Helper Process needs to execute a GET request to the unbounded BDLO. This is different in the Smart BDLO algorithm, which has instant access to the ledger, since it works as an unbounded BDLO and a helper process at the same time.

With that said, if we exclude the GET request of the Helper Processes, the performance should be identical with each other. So we chose as an evaluation metric the overhead of the GET requests that a Helper Process executes for a different number of records. To measure the overhead, we copied the existing Helper Processes implementation and modified in order to find the time it takes for a GET request to complete. We used the libraries Instant and Duration to calculate the time from the moment the GET request is sent until it is executed. Figure 5.25 shows the added modification that was appended on the code shown in Figure 4.10.

```
while(true){
                start = Instant.now();
                List<Record> list =  GetLedger(calculateKey());
                end = Instant.now();
                Duration duration = Duration.between(start, end);
                // MORE CODE ...
}
```

Figure 5.26 : Added Modification that measures the time a GET request needs to be completed

For the evaluation we used the testing environment of Emulab [6] with the the topology seen in Figure 5.27. This is the architecture of the Helper Processes algorithm that we saw before. As we can observe, we used 4 Helper Processes and each BDLO had 5 servers. The type of all nodes is emulab-xen. Emulab-xen is essentially a virtual machine (or virtual nodes) which runs a selected operating system. For our virtual nodes, we used Ubuntu 18. Additionally, all the virtual nodes have routable IP address, which allows the instances to communicate through the Internet. Each node has at least 1GB of RAM, maximum 16GB of virtual disk.

For the testing we programmed the Helper Processes, to measure the amount of time a GET request would take to complete, in milliseconds, for different number of records. The number of records changed by modifying the server code to 5000 new records every 10 GET requests it receives. Figure 5.28 presents the results of the testing. We can see a steady increase of the overhead while the number of records also increase. The number of records, is initially set to 5000 and and increases by 5000 till it reaches 100000. For each number of records, the Helper Processes executed a GET request of total 10 times, and we calculated the average time needed for completion.

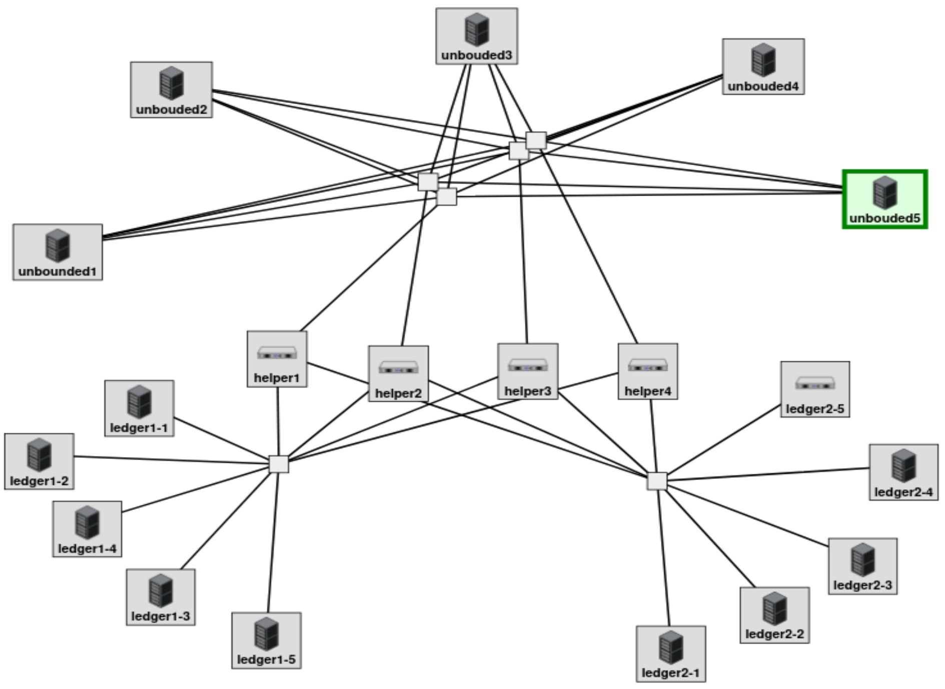


Figure 5.27 : Topology of Helper Processes implementation used in Emulab

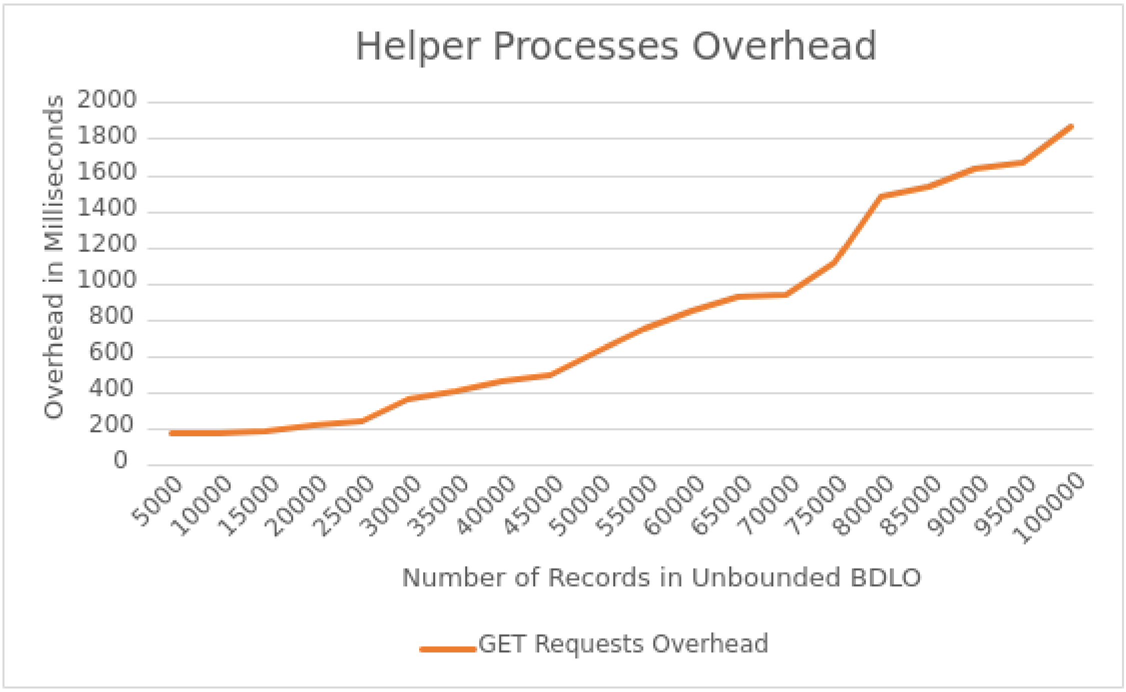


Figure 5.28 : GET Requests Overhead in Helper Processes for Different Number of Records

The results show that the overhead for 100000 records, is on average 1,8 seconds. This is relatively a small number, however, blockchains nowadays tend to have millions and millions of records, which in our scenario could have a very negative effect. This shows that the Smart BDLO implementation could be a better choice for solving the Atomic Appends problem. However, the Helper Processes algorithm might suit a project better than the Smart BDLO. For example, when we don't have access to the code of a BDLO and cannot modify the servers in order to execute the Atomic Appends. We could use the Helper Processes implementation just by modifying it to read the records from the particular BDLO. Of course, for this to work, the BDLO must at least have an API which allows clients to read and append records on the ledger.

Chapter 6

Conclusion

6.1 Summary

After studying various articles about Asynchronous Byzantine Distributed Ledgers, and the Atomic Appends problem, we decided to extend Andreas Chrysanthous work, and implement the Helper Processes and Smart BDLO algorithms as presented in Chapter 3. We first developed the bounded version of a BDLO which was necessary in order to implement the two algorithms. Then, we continued with the implementation of the Helper Processes algorithm which was the longest task in the project, due to the modification that had to be made in order to make it work as it should. Afterwards we continued with the Smart BDLO algorithm. This was much more easier to develop since a lot of its code was already done for the previous algorithm. Finally, we tested the overhead of the Helper Processes by computing the average time of a GET request to be completed, for different amount of records that were stored in the unbounded BDLO.

6.2 Problems Encountered

The first problems we encountered, was with the Record class. In the BFT-Smart library, all requests need to be converted into bytes first. However, in Java, in order for a class to be converted into bytes, it needs to implement the Serializable class. We didn't know this at first, but after some help from the stack overflow community [7], we managed to resolve this issue. Additionally, in our implementations, we used HashMaps to store records. For efficiency we used the record to be the key, which is what the Hashing Function takes as an input to generate an index for the underlying array. In Java, when a variable is used for Hashing it uses the default HashCode method which is provided by the class Object. But in this way, the HashCode of two records that had the same value, produced two different hashes, which is not what we needed for the implementation. We resolved this by implementing a hashCode method, in the class Record, so that it would take as input the values of the records.

Another problem we faced was when we had to implement a way for an instance to be able to communicate with more than one BDLO. This took a lot of time and effort to solve, because we had to interfere with the BFT-Smart code, which is low level programming. We solved this by adding the necessary methods that allowed an instance to be able to read a set of IPs from other file than Hosts.config, and connect to them. More details were given in Section 4.5.

Furthermore, while testing the methods mentioned above, we encountered another problem. When two instances connect to the same BDLO, we need to give them different IDs when launched. For example, if we launch four Helper Processes, they must have IDs 0,1,2,3 respectively and they connect to every BDLO. The clients also connect to all the BDLOs, so they need to have IDs that start from 4, in order to avoid communication errors between the instances.

Lastly, we realised that the port of a client changes for every request. At first we wanted to let the clients know when an Atomic Appends operation was completed, but due this problem, we decided to let the clients execute GET requests to the bounded BDLOs.

6.3 Future Work

In this study we considered a specific version of the Atomic Appends problem, where only two clients exchange records with each other and there are only two DLOs. This problem can be generalized to *k*-Atomic Appends, involving *k* clients with *k* records and up to *k* DLOs. So a next step could be to implement an algorithm which solves the *k*-Atomic Appends. Our implementation could be used for the back-end but it would need to be modified in order to find the *k*-matching records. Additionally the frontend (GUI) code would need to be reimplemented so that a client can append its record with the correct structure.

Additionally, a program could be implemented that configures the configuration files automatically. The manual configuration is mistake-prone, and takes a lot of time to setup for a big number of servers and clients.

In this study, we implemented a prototype. In the future, a system like ours could be used in production by real clients, but a lot of things need to be done in order to reach that state. Firstly, the system needs to be implemented on a modern programming language, since Java is outdated and slow. Also, a user interface that can be accessed on a browser (web application) must be developed. There, the clients would be able to exchange digital goods with each other, see previous transactions and view the records of any ledger they desire. Another challenge for this project would be to integrate the various ledgers (i.e. Bitcoin ledger, Ethereum etc.) within the system, in order to execute the requests correctly. This would be ambitious because there is a plethora of ledgers currently in production and the API of the ledgers, that would allow the clients to execute requests, might differ from each other. However, this problem could be avoided if the system only supports a number of ledgers. For example, a system that would specialize in the trade of a cryptocurrency and a car ownership. In this case, only two ledgers would have to be integrated within the system, the ledger of the chosen cryptocurrency and the ledger of car ownership. Lastly, it is crucial for a system like this to have robust security measures in order to protect its clients from hackers and system failures.

Bibliography

[1] Java Programming Language : https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html

[2] Andreas Chrystanthou "Implementation of DLO in a Byzantine Fault Tolerant System, with the use of the tool BFT-Smart", Diploma Project, Dept. of Computer Science, University of Cyprus 2022.

[3] Cholvi, Vicent & Fernández Anta, Antonio & Georgiou, Chryssis & Nicolaou, Nicolas & Raynal, Michel. (2020). Appending Atomically in Byzantine Distributed Ledgers. https://www.cs.ucy.ac.cy/~chryssis/pubs/CFGNR-EDCC20.pdf

[4] Maurice Herlihy. Atomic cross-chain swaps. In Calvin Newport and Idit Keidar, editors, Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018, pages 245–254. ACM, 2018.

[5] Bessani, Alysson & Sousa, João & Alchieri, Eduardo. (2014). State machine replication for the masses with BFT-SMART. Proceedings of the International Conference on Dependable Systems and Networks. 355-362.10.1109/DSN .2014.43.

[6] Emulab: https://www.emulab.net/portal/frontpage.php

[7] Stack Overflow : https://stackoverflow.com/

[8] Thorsten Linz. The Five Ways Blockchain Will Redefine Car Ownership, March 13 2018, Move Forward Blog, https://medium.com/move-forward-blog/the-five-ways-blockchain-will-redefine-car-ownership-f2acb5568c77

[9] Fernández Anta, Antonio & Georgiou, Chryssis & Konwar, Kishori & Nicolaou, Nicolas. (2018). Formalizing and Implementing Distributed Ledger Objects. In SIGACT News, 49(2):58-76, June 2018.

[10] Antonio Fernndez Anta, Chryssis Georgiou, and Nicolas Nicolaou. Atomic appends: Selling cars and coordinating armies with multiple distributed ledgers. In International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2019), pages 39–50, Paris, France, 2019.

[11] Haitham Nobanee, Nejla Ould Daoud Ellili, Non-fungible tokens (NFTs): A bibliometric and systematic review, current streams, developments, and directions for future research, International Review of Economics & Finance, Volume 84, 2023, Pages 460-473, ISSN 1059-0560, https://doi.org/10.1016/j.iref.2022.11.014.

[12] Zheng, Zibin & Xie, Shaoan & Dai, Hong-Ning & Chen, Xiangping & Wang, Huaimin. (2017). An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends. 10.1109/BigDataCongress.2017.85.

[13] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. ACM Trans. Program. Lang. Syst. 4, 3 (July 1982), 382–401. https://doi.org/10.1145/357172.357176

[14] Vincent Gramoli, From blockchain consensus back to Byzantine consensus, Future Generation Computer Systems, Volume 107, 2020, Pages 760-769, ISSN 0167-739X, https://doi.org/10.1016/j.future.2017.09.023.

[15] BFT-Smart Library on GitHub : https://github.com/bft-smart/library

[16] L. Rodrigues and M. Raynal, "Atomic broadcast in asynchronous crash-recovery distributed systems," *Proceedings 20th IEEE International Conference on*

Distributed Computing Systems, Taipei, Taiwan, 2000, pp. 288-295, doi: 10.1109/ICDCS.2000.840941.

[17] Wright, Craig S, Bitcoin: A Peer-to-Peer Electronic Cash System (August 21, 2008). Available at SSRN: https://ssrn.com/abstract=3440802 or http://dx.doi.org/10.2139/ssrn.3440802