Geo-located dynamic replication in Legion

Frederico Pinto Aleixo
fp.aleixo@campus.fct.unl.pt

NOVA University of Lisbon,
Faculdade de Ciências e Tecnologia, Almada, Portugal

Abstract. With the rise of mobile applications such as Waze and Pokemon GO, the greater focus towards social and collaborative features in modern applications is hard to ignore. In such applications, users' interest lay on what is around their immediate surroundings, as opposed to what is happening in areas distant from their current location. Legion is a system that reduces latency and relies less heavily on servers, by enabling client devices to communicate with one another in a peer-to-peer fashion. However, clients are forced to share all application data with one another. While this improves user experience, it makes clients keep data that may not be relevant to them, especially if such data is in some way related to their geographic position. In this case, clients should not have to receive information that is too distant to be of any relevance to them. We aim to address this, making it so that clients only keep data that is of their interest, in other words, that is close to their geographic position.

Keywords: Peer-to-peer · Location-based replication · Mobile applications · Legion · SAMOA.

1 Introduction

As modern applications shift more and more towards a standard of greater interaction between people, meeting the new demands that arise with this paradigm shift is essential to captivate users and keep them engaged. One such demand is that applications become more responsive and sturdy, allowing for quick responses to user input and continued use after the servers become unavailable.

For applications that are implemented using a centralized infrastructure, not only is this an almost guaranteed impossibility, but such approaches present additional inconveniences. By allowing servers to mediate all user interactions, they become a scalability bottleneck. As more and more clients connect to it, the amount of work that has to be done by a server to not only answer all incoming user interactions, but also to make sure that other clients become aware of such interactions, increases in a polynomial fashion. On top of this, users that are close to one another are forced to communicate via the server, which often leads to a considerable increase in latency. To mitigate this situation, Legion [1] was created, relying on direct interactions between clients, therefore making the system less dependent on a centralized infrastructure and, consequently, reducing

latency and avoiding potential bottlenecks. However, because Legion forces users to exchange all application data, they will inevitably receive information that is potentially useless to them, should that information be related to a geographic position that is simply too far away to be of any interest.

Our aim is to improve the framework further so as to better cater to the needs of modern-day users, by enabling them to receive data based on their current geographic position, much like what is being done with applications such as Waze, Pokemon Go and Foursquare. We hope that, by doing so, the amount of unnecessary data traded between peers decreases, thus expanding the benefits of Legion as a peer-to-peer replication mechanism.

To do this, a new protocol was developed. It allows for users to connect to other nearby users and exchange object information with them, while ensuring that any data received pertains to objects that are physically close to the location of the receiver. As a client moves, objects that are too far away are discarded and connections to distant peers are closed. By the same token, connections to new peers are formed and information pertaining to nearby objects is acquired, which is sent to the client either via the server or via that client's peers themselves.

Our protocol, much like what had already been observed in previous tests [1] with Legion, has shown to have lower latency than a client-server model. On top of that, it also performs better than the original protocol of the framework under certain conditions, specifically when the interactions between users and objects are plentiful.

In the following pages we will explore how this solution came to be, what sort of challenges and problems shaped it into what it is, and how it compares to the client-server and original Legion models.

2 Related work

In this chapter, we survey the core concepts directly associated with this work and compliment with some analysis on the state-of-art in the relevant fields.

2.1 Causality

Causality pertains to how the execution of an operation may affect the execution of other operations in a given environment, as well as be dependent on operations that came before it. To better understand this concept, we will start by taking a look at causal relations between operations, and then we will try to understand what causal consistency [2] is and how we can achieve it.

Causal relations Causal relations can be observed when the operations taking place in a system are potentially causally linked. In other words, if we have two operations x and y, then x can be the cause of y if and only if they both occur in the same node and x was the first to execute, or, in the event that they execute on different nodes, if y is informed about the execution of x via some message received from a node that knows about x.

Let us take a look at the three general scenarios that can happen when it comes to causal relations between operations. Edges from x to y indicate that y depends on x:

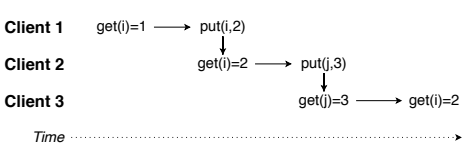

Fig. 1: Example of causal relations between operations

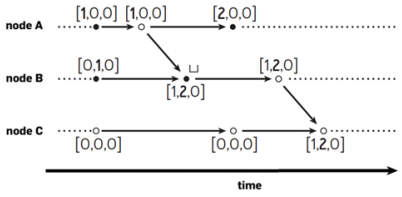

Fig. 2: Version vectors (from [3])

1. If x and y are two operations in the same node, then y depends on x if x happens before y. We can see an example of this in Client 1 of Figure 1, where put(i,2) depends on get(i)=1.
2. If x is a put operation and y is a get operation that returns the value written by x, then y depends on x. This can be seen in the edge connecting the put(i,2) operation in Client 1 with the get(i)=2 operation in Client 2.
3. For operations x, y and z, if y depends on x and z depends on y then z depends on x. This transitive property allows us to see that get(i=2) in Client 3 depends on put(j,3) in Client 2.

Causal consistency Causal consistency guarantees that an operation *o* is only executed in a replica after all operations that happened before *o* have been previously executed. There are various different manners of enforcing causal consistency. One such manner would be to use version vectors [2], where an operation's dependencies are sent with the operation itself. This way if, upon reception, all dependencies are not satisfied, the node will already have the necessary knowledge to determine which operations are missing. Figure 2 illustrates how version vectors work.

If we take a look at node B for instance, we can see that it has a vector of [0,1,0]. It then receives a message from A, and with that message it is able to determine which operations A has already executed. It can then take its vector, a new operation it executed, and the [1,0,0] vector received from A and merge them into a new vector that contains all that information, [1,2,0]. Note that version vectors will only record important events (such as writes), which is why the next operation in B is still [1,2,0] instead of [1,3,0].

2.2 CRDTs

CRDTs [4] (Conflict-Free Replicated Data Types) are data structures that allow for guaranteed eventual consistency, through the usage of replicas. A replica is

free to execute an operation without having to immediately synchronise with other replicas, as the operation is eventually sent in an asynchronous manner to those replicas. All replicas end up applying all the updates (which may or may not be applied in different orders), thus preserving consistency.

There are three different styles of synchronisation used by CRDTs:

- *State-based* (passive) replication, in which the system transmits the state of the updated source replica to other replicas, to propagate changes. The receiving replicas will merge their current state with the one they receive.
- *Operation-based* (active) replication, where the system transmits operations that mutate the state of the source replica (these operations are known as updates). In order to achieve this, every update must reach the causal history of every replica eventually.
- *Delta-based* replication, where state changes taking place in a replica are stored in a delta, which is then sent to other replicas so that they can update their own state with the received changes.

CRDTs support numerous already existing data structures, such as Counters, Registers and Sets (from which Maps, Graphs and Sequences are derived).

2.3 Vector-Field Consistency

Vector-Field Consistency [5] (VFC) allows to strengthen or weaken replica consistency. Using distributed multiplayer games as an example, the consistency of the data may may change depending on the current game state.

The consistency of any given object depends on that object's distance from a pivot. The pivot has a position in the virtual world, which can change over time. Object consistency is determined through consistency zones, which are concentric areas generated around pivots in such a way that objects positioned within the same consistency zone have to abide by the same consistency degree.

In short, VFC allows programmers to express the consistency requirements that their applications need easily. It is also noteworthy that they are widely applicable, not being restricted to game development.

3 Algorithm for geo-partial replication

In this section, we will examine the various parts that make up the protocol in detail, explaining what hurdles justify their existence and what role they play in the final solution.

3.1 System model

In our system, we assume that clients communicate over a fixed infrastructure (5g, Wi-fi), i.e., nodes do not communicate directly in an ad-hoc fashion. Both the signalling server as well as the objects server are running in the cloud. A set

of objects exists, which are replicated by the nodes in the system. Geographic positions are associated to both objects and nodes, with nodes only being interested in objects that are physically close to their position. Lastly, certain objects are replicated in every node, similarly to what happens in Legion's base model.

3.2 Background

Legion is a framework that enables client web applications to replicate data from servers using delta-based CRDTs [4], as well as synchronize these replicas among them. Unlike systems that cache objects at the client-side, it allows clients to synchronize directly among each other and with the server, using a peer-to-peer model. To enable this, some clients are chosen to act as bridges between the server and the other clients (these chosen clients are named active nodes). These nodes are responsible for uploading updates executed by their peers and downloading new updates that were executed by certain clients that did not establish a direct connection to other peers, having only connected to the server. We will now explore how some of these connections work in more detail.

Legion relies on two servers, one that mainly keeps track of all data structures in use (named *ObjectsServer*), and another that focuses mostly on tasks such as authenticating and managing client connections (named *SignallingServer*). Let us take a look at Fig. 3 to understand how a network in Legion is formed.

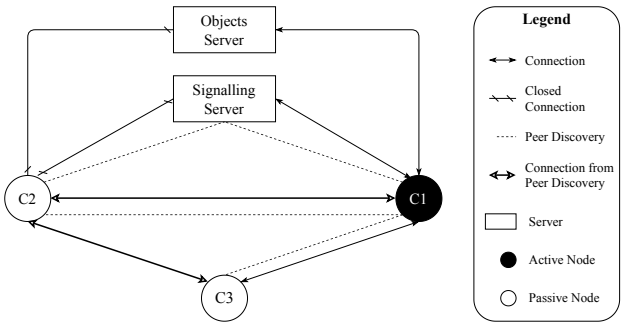

Fig. 3: Legion overlay example

Initially, client (or node) C1 is connected to both servers. C2 then connects to the network, and at this point both C1 and C2 would be in an active state, with C2 also being connected to the servers. Through peer discovery, C2 discovers C1, and establishes a P2P connection with it. A bully algorithm will then choose which one of these two will remain in the active state, eventually picking C1 since it has a lower ID. C2 then becomes a passive client, closing its server connections. Say client C3 was directly connected to C1. Using peer discovery, C3 can establish a connection with C2 via C1, since C1 is already connected with C2. Thus, the three nodes are able to communicate with one another yet only C1 actually needs to interact with the servers.

3.3 Partial replication

If we are to make it so that clients only receive data that is pertinent to them, the full replication model that Legion uses had to give way to a partial replication approach. This is because, with Legion in full replication mode, when a peer first connects to another, it goes through a synchronization process. This process involves an exchange of all data structures stored locally at each peer, as well as all updates they may have over said structures. This process guarantees that either peer is now entirely up to speed on everything the other one knows, but it also means that they were forced to store data that may not have been pertinent to them.

Addressing the challenge presented above required some changes to the synchronization phase. Now, when a peer receives data sent by another peer, one of two things may happen. If our peer is the objects server, then it is in the server's best interest to keep all the received data, just as before. However, if our peer is a client, then all it will do with the data received is determine if there are any objects that it already has locally stored. If so, he may send a record of all known updates of said data structures to the peer. In turn, the peer will do the same, so both of them end up with the same version of the data. Additionally, each client keeps a record for every neighbour, on its interest in what structure, so that any information sent after the synchronization phase will undoubtedly be relevant to the receiver, ensuring that no client is ever required to store or receive unnecessary data.

3.4 The GeoLocOverlay

Given that nodes follow a peer-to-peer replication model and that they are interested in objects which are close to their geographic position, it is only logical for the overlay they use to communicate to be associated with their location, so that the replication only takes place in nodes that are interested in those objects.

In order for nodes to take positions into account when forming connections with one another, a new overlay had to be designed. The initial idea was for any peer who connects to the overlay for the first time to send a message to the signalling server with its current position, so that the server could examine each active node (and their position, which it would keep stored) and determine the two closest active nodes to the new node. Then, it would send a message to each of them with the position of the new peer, so that they could keep propagating this message to the peer that is closest to the new node. When the message reached a node that was close enough to the new peer, it would stop being propagated and the two nodes would connect. Then, it would be a matter of sending the location of our other adjacent peers to the new peer so that he would find out about other nodes in the vicinity and decide to which of them it should connect to.

While this approach seemed reasonable at first, it had two shortcomings. The first was that the number of messages that would be propagated throughout the overlay would explode, severely affecting the efficiency of the protocol. Secondly,

there could be a scenario where, despite two nodes being close enough to one another, they would be unable to realize that and would never connect. Figure 4 exemplifies this.

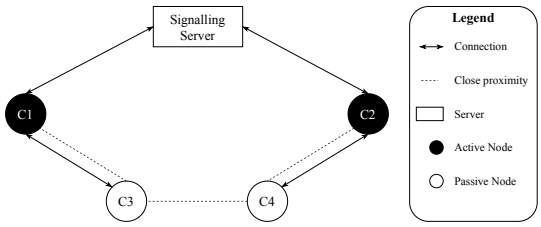

Fig. 4: Proximity detection failure

We can look at the figure as if we had two separate branches of connected nodes, the left branch which contains nodes C1 and C3, and the right branch with C2 and C4. As we can see, the nodes at the end of each branch are close enough to one another to be able to form a connection, yet it is impossible for them to do so because they will never acknowledge each other. C3 will only ever know the positions of its peers and, at most, of the peers of its peers, which means that the only node that it knows the position of is C1. A similar situation is happening in C4, with C2 being the only node it knows the position of.

Because relying so heavily on nodes to decide what connections should be formed had shown to be a rather naive approach, it was decided that the server should have a more prominent role in this. If the server keeps a record of all node positions, then nodes can easily contact the server to discover new peers and receive an answer directly from it, without the need to propagate nearly as many messages throughout the overlay as before and without causing the disconnected branches scenario to ever happen.

The final overlay (named *GeoLocOverlay*) keeps all nodes directly connected to the signalling server. The pseudo-code for the overlay can be found below. When a new node joins (line 7), it will send its position to the server, and in turn, the server will return with a list of all nodes that are physically close to it (line 66). When a node moves a certain amount of units since the last time it updated its position, it sends a message to the server with its new position (line 57) and, once again, receives a list of nodes that are nearby (line 66). It also informs its current peers of its current position (line 59), so that they do not need to wait for the next server update to register this new information.

Since this protocol is not intended to be uniquely dependent on a central server, should a node become disconnected from the signalling server, it will send its new position to its peers (line 54), so that they may acknowledge its location and also propagate it to their neighbours. This case is of particular interest, where a node moves to a distinct location, and nodes that in the logical level are distant, upon realising that a new node is close by, become interested in forming

a connection with it. This allows the node to keep forming new connections even when it is not connected to the server (line 37), but as we have previously seen, without the help of the signalling server there are some connections that the node will be unable to form on its own.

Periodically, nodes will remove peers that are too far away (line 16). And, in case a maximum number of connections hasn't been reached, the protocol dictates that nodes try to connect to other nodes of interest in the neighbourhood, making them peers (line 19).

Nodes have two distinct ways to realise which of their neighbours are of interest: they can either make use of the server messages containing the IDs of all nearby nodes (line 10), which are received every time an update is sent to the server, or receive a new node's position via the messages sent from their neighbours, when the signalling server is unavailable (line 37).

This approach solves the issues we previously had, and although the number of messages that pass through the network is still considerable, it is nowhere near as high as it would be using the first protocol.

```
1  GeoLocOverLay {
2    let lastSentPos = currentPos; //Last sent position to server & peers
3    let peersOfInterest = new Set();  //Peers that are close to our position
4    const MAX_DISTANCE = 20; //Maximum distance allowed between connected peers
5    const MAX_PEERS = [1 to 10]; //Max no. of peers that can be connected to us
6
7    function onServerConnection() //Send our position to the server so that it
         can tell us what our nearby peers are
8      serverConnection.send(currentPos);
9
10   function onServerResponse(message)
11     for each peerID in message.peerIDs
12       if(!ourPeers.contains(peerID))
13         peersOfInterest.add(peerID);
14
15   function updateConnections()//Every 5000ms or so update our peer
         connections
16     removeDistantPeers(); //Disconnect from peers that are too far away
17     if(peerCount < MAX_PEERS)
18       let remainingSlots = MAX_PEERS - peerCount;
19       /** For each ID in peersOfInterest, send it a join request and
             decrement remainingSlots. If remainingSlots == 0, stop */
20     if(peerCount > MAX_PEERS)
21       removeMostDistantPeer();
22
23   function onPeerJoinRequest(message)
24     if(peerCount < MAX_PEERS && !ourPeers.contains(message.sender))
25       let dist = distance(message.senderPos, currentPos);
26       if(dist <= MAX_DISTANCE)
27         connectPeer(message.sender);
28       else
29         peersOfInterest.delete(message.sender);
30
31   function onPeerConnection(peerConnection) //When we connect to a peer
32     peersOfInterest.delete(peerConnection.ID);
33     toPropagate = true; //Enables our peers to send our position to other
         neighboring peers in case the server becomes unavailable
34     peerConnection.send(currentPos, toPropagate);
35
36   function onPeerPosition(message)
37     if(!message.toPropagate) //Receiving pos info of a new node (only true
         when the server is unavailable)
38       /** If the pos is close enough, add the node to our peersOfInterest*/
39     else  //Receiving pos info from one of our peers
```

```
40        updatePeerPos(message.sender, message.pos);
41        sendPositionToPeers(message);
42
43    function sendPositionToPeers(message)
44      if(message) //If we're propagating a peer's position
45        if (/**we aren't connected to the signalling server*/)
46          message.toPropagate = false;  //Acts as a TTL of 1 to propagate the
                message through a single logical layer
47          let except = message.visitedIDs;
48          sendToPeers(message, except);
49      else
50        let visitedIDs = [];
51        for each peerID in peers
52          visitedIDs.push[peerID];  //Let our peers know that we have already
                sent the message to these peers
53        let toPropagate = true;
54        let message = generateMessage(currentPos,visitedIDs,toPropagate);
55        sendToPeers(message);
56
57    function updatePosition() //Every 5000ms or so
58      if( distance(lastSentPos, currentPos) > MAX_DISTANCE)
59        /** Send our new position to our peers and the server*/
60        lastSentPos = currentPos;
61  }
62
63  ServerSide {
64    let nodesPos = new Map(); //Key: client ID; Value: node pos
65
66    function onNodePos(client, pos)
67      nodesPos.set(client, pos);
68      //** for each node in nodesPos, if it is close enough to the client, send
            its ID to the client */
69
70    function onClientDisconnect(client){ nodesPos.delete(client) }
71  }
```

3.5 The Objects Bully

Now that peers are able to establish connections amongst each other based on physical proximity, we need to ensure that the bullies who act as bridges between the objects server and their peers are properly distributed.

Using the current protocol to achieve this, because it relies solely on the ID of nodes in the overlay to compare them and decide who the bullies are, the nodes with the lowest IDs are automatically the best option. While this protocol was more than sufficient when all clients were interested in all objects, now that clients are most often only propagating a small subset of all available objects, it can result in some undesirable situations. To exemplify, it could happen that the designated bully in a specific group of nodes isn't interested in the entirety of objects that the group is. This implicates that information regarding objects that the bully is not interested in would never be sent to it (since as we have seen, data will only get sent to a peer if it belongs to an object that it is interested in), and therefore would never reach the objects server. This would nullify any attempt to safely store the new state of that object in the server.

To mend this situation, a new bully protocol was created. This new protocol works slightly differently from the previous one, in the sense that nodes no longer have just a single bully attributed to them, but instead have a bully per object they are currently interested in. The idea behind this concept is that, if for every

object a node is currently propagating, it is able to find a peer that is not only interested in that object, but also has an open connection to the objects server, then it can designate that peer as the bully for that specific object. If a node is able to do this for every object it is propagating, then it knows that it no longer needs to keep a connection to the objects server open, since it can rely on its peers to relay the information it sends regarding those objects to the server. Figure 5 illustrates this more clearly.

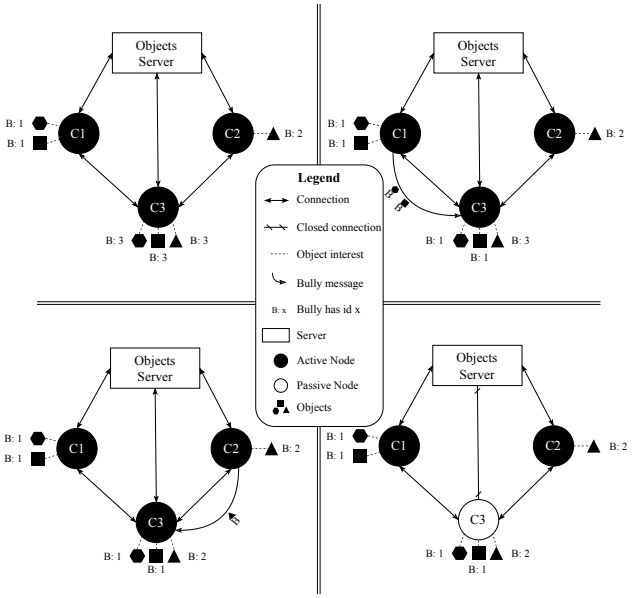

Fig. 5: Objects bully

Reading the image from top to bottom, left to right, in the first scenario we can see that all nodes are connected to the ObjectsServer, and for each object they are interested in (depicted by the smaller geometric shapes) they have set themselves as the bully.

In the next scenario, node C1 sends a bully message to C3 letting it know that it is currently a bully for the circle and square objects. Upon receiving this message, C3 will compare its current bully for the circle and square with C1, using the same principle as before in which the bully with the lowest ID is the best one. As C1 has a lower ID than C3, C3 updates the bully for those objects, setting the node with ID 1 as the new bully.

Moving to the bottom left, a similar event takes place, in this case with C2 sending a message to C3 telling it that it is currently being a bully for the triangle object. C3 compares both IDs once more, determines that C2 is a better bully than itself, and so sets C2 as the bully for the triangle.

Lastly, C3 has now managed to define a bully other than itself for each of its objects, which means that it is safe for it to close its server connection, since it knows that any information sent to its peers regarding the objects will be relayed to the ObjectsServer by them.

With the new bully protocol implemented, we can now ensure that any node executing operations on any object will be able to send those changes to the objects server, whether via a direct connection to the server or via a peer that is connected to it. By the same token, if a node arrives at a location where there is an object of interest for which there is no bully among the node's peers, the node itself will connect to the objects server (if it was not connected already) and in this manner it is able to obtain the object and interact with it, even if none of its peers were doing so. The pseudo-code for this protocol is the following:

```
1  ObjectsBully{
2    let bullies = new Map(); //Key: objectID; Value: bullyID
3    let objectIDs = objectStore.objectIDs;
4
5    for each objID in objectIDs //On startup, we are the bully for all objects
6      bullies.set(objID, thisNode.ID);
7
8    function floodInterval() //Every 8 seconds or so let our peers know the
         objects that we are bullies for
9      for each ID in objectIDs
10       floodBully(ID);
11
12   function floodBully(objID)
13     if (/**we have no bully for objID or if we're the bully*/)
14       //Message peers interested in objID that we're bullies for it
15
16   function onMessage(peerID, peerObjID, connection)//On bully msg
17       if (/**peerID <= ID of our current bully for object peerObjID OR no
              bully is defined for object peerObjID*/)
18         bullies.set(peerObjID, peerID); //Make peerID the new bully
19         setBullyTimeout(peerObjID);
20       else
21         if we are a bully for peerObjID AND peerID > myID
22           //Reply to peerID so he knows we're a better bully for objID
23
24   function onClientDisconnect(peerConnection)
25     for each objID in objectIDs
26       if(peerConnection.ID == bullies.get(objID))
27         if objID is in our possession
28           bullies.set(objID, myID);
29         else
30           bullies.delete(objID);
31
32   function onServerConnect()
33     for each objID in objectIDs
34       if (/**we have no bully for objID or we're the bully*/)
35         floodBully(objID);
36
37   function setBullyTimeout(objID)
38     /**After the timeout, become the bully for objID again*/
39 }
```

4 Evaluation

To test the correctness of the new protocol, initial tests consisted of simulating several simple scenarios in which nodes would be placed in specific or random

positions, with the goal of observing how they conducted themselves when it came to connecting to new nodes, disconnecting from far away nodes and trading object information between one another.

We then advanced to a more interesting testing scenario, one that would attempt to emulate an app such as Foursquare [6]. Foursquare allows users to discover information about nearby locations, enabling them to perform a *"Check In"* to obtain a list of nearby places. Users may also use the *"Explore"* feature to search for a specific type of location using different categories, as well as leave a review after visiting a place. Our application works in a similar manner to some degree, with clients receiving data on nearby objects and being able to interact with them. Depending on the test, these objects can either be CRDT counters that simulate some sort of like/dislike counter, or CRDT maps where strings are stored, to mimic leaving a comment or review.

To simulate the positions of both clients and objects, we made use of two datasets. One of these datasets contains the geographical coordinates of several taxi trips in the city of Porto, while another has the positions of bus stops in that same city. The first dataset will be used to simulate client movement, and the second dataset will be used to choose potential object positions. With the information from these datasets, we will use three different overlay models to conduct our tests: the GeoLocOverlay in partial replication mode (*glo-partial*), in full replication mode (*glo-full*), and the client-server overlay in partial replication mode, which will disable any sort of peer-to-peer interaction. We aim to answer the following questions:

1. How is the total number of messages sent to and from the signalling and objects server affected when we compare the client-server model and the initial full-replication Legion with the partial replication model?
2. How is the total number of bytes sent to and from the signalling and objects server affected when we compare the client-server model and the initial full-replication Legion with the partial replication model?
3. What are the maximum and minimum number of messages sent by any given node in each of the three different scenarios? On average, how many messages are sent by nodes?
4. What are the maximum and minimum number of bytes sent by any given node in each of the three different scenarios? On average, how many bytes are sent by nodes?
5. Is there an improvement in latency when we use the GeoLocOverlay in place of the client-server overlay?

We created scenarios with 5 clients moving through routes of 40 positions, extracted from the taxi trips dataset. The selected routes are close to one another to ensure clients can connect to each other at some points. Additionally, 50 objects are also created, with their positions taken from the bus stops dataset. These positions were picked in a manner that ensures that, for any object, a client will interact with it at some point. Some clients share objects for a certain period of time, while other clients are the only ones interacting with certain objects.

4.1 Like Counter

In this scenario, the created objects are CRDT counters. Clients will increment all nearby counters by one every three seconds. When a client approaches a counter, it receives the current state of the counter either via synchronising with its peers or, if no peers are interested in that object, via the objects server. When a client moves too far away from a counter that it was interested in, the counter is removed from that client's local CRDT storage.

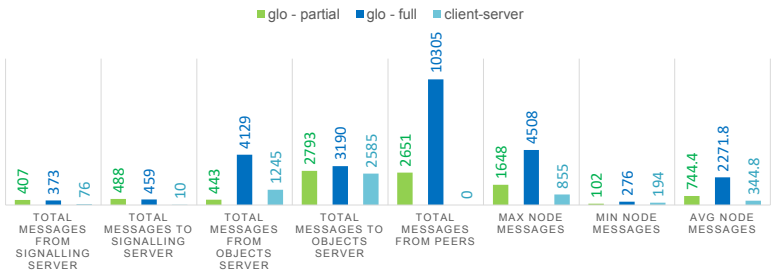

Fig. 6: Like counter - message number

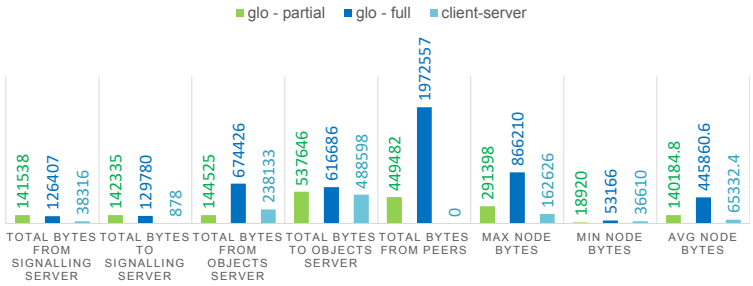

Fig. 7: Like counter - byte number

As can be seen in figure 6, using the new overlay with partial replication results in an overall lower number of messages traded between nodes and the objects server and amongst peers when compared to the full replication model. While the client-server model (in partial replication mode) still manages to have slightly less messages sent to the objects server than the glo-partial model, the number of sent messages from the objects server has been reduced significantly, not only because not all peers require to keep an open connection to the objects server as they do in the client-server model, but also because they will only receive data that is relevant to them, unlike what happens in the full replication model.

What can be seen in the figure 7 mimics what the previous figure, albeit at a different scale. This is because the many operations that are executed on each object (which are increments of one) do not require a lot of bytes to be sent at a time. In the next scenario we will see that, because we have much fewer operations but each of them involves a higher amounts of bytes, exchanging fewer messages does not necessarily translate to exchanging fewer bytes.

In summary, when it comes to communicating with the objects server, the glo-partial model has come out on top. For everything else, the client-server model manages to be the one that exchanges less messages and bytes.

4.2 Post Comment

The created objects in this scenario are CRDT maps. The first time a client is close to a map, it will write to it using a randomly generated 10kb string. The process of synchronising and discarding object data is identical to that of the previous scenario.

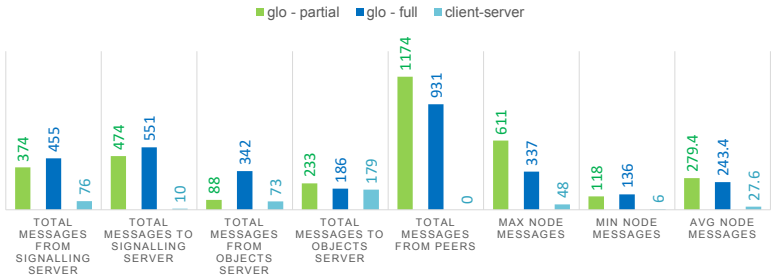

Fig. 8: Post comment - message number

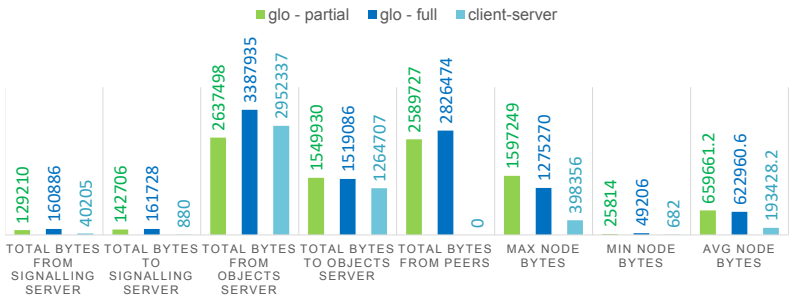

Fig. 9: Post comment - byte number

Examining figure 8, we can see that this time around the number of messages traded between peers and sent from peers to the objects server was greater in the glo-partial model than in the full replication one. We can understand why this may be when we take into consideration that, unlike in the previous scenario where a given node could interact several times with an object, in this scenario each node will only interact with each object once. This means that a good portion of the messages traded between peers were most likely bully and peer discovery related messages as opposed to messages containing data on the operations executed at each map.

It is interesting to note that, in figure 9, the partial model seems to fare much better. If we compare the total messages received from peers with the number of bytes received from peers for instance, despite peers sending more messages to one another in the partial model, the total number of bytes sent is actually lower than that of the full replication model.

To summarize, the client-server model has proven to send the least messages and bytes in practically all interactions of all three models. However, comparing the glo-partial model with the full replication model, the number of overall exchanged bytes was still lower.

4.3 Latency Tests

In this scenario there is a single CRDT map shared between all clients. Every 10 seconds clients write to this map, and that write is propagated through the overlay. As soon as a node receives a write, it prints the time elapsed since the operation was executed in the original client until it was received. To more realistically simulate what would happen in a real life situation, artificial delays were introduced. The values obtained represent the average latency of an operation registered by a client.

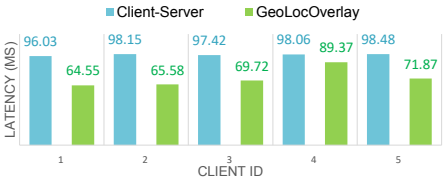

Fig. 10: Lower latency

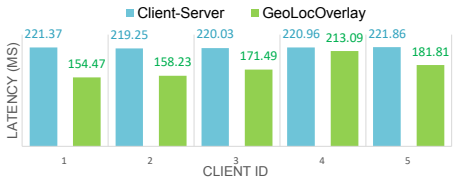

Fig. 11: Higher latency

In the first test, a delay of 5 milliseconds between peer to peer interactions and 40 milliseconds between peer-server interactions was considered. These values were changed to 40 milliseconds and 100 milliseconds for the second test.

Both figures show that using the GeoLocOverlay results in lower latency, which can be explained by the fact that nearby nodes are now able to communicate directly, without having to use the server as an intermediary.

5 Conclusions

The objective of creating a protocol that enables clients to receive data based on their current geographic position was accomplished. By making use of the new overlay and bully algorithm, we ensure that clients are able to receive information regarding nearby objects in a correct and efficient manner. We also guarantee that, in the event that no clients are propagating an object, that object's data is safely stored in the objects server, so that the next client that propagates said object may synchronize with it and obtain that object in its most recent state.

The obtained results have shown a considerable overhead associated with nodes communicating their position to the signalling server and obtaining from it the peers that are closest by. However, the number of messages sent to the objects server saw only a slight increase, and the number of messages from that same server was significantly reduced. Despite the fact that the client-server model still remains the lightest of the trio when it comes to both messages as well as bytes exchanged, the partial replication model using the GeoLocOverlay has proven to not only offer reduced latency when compared to this one, but also to exchange significantly less bytes and messages than the full replication model in situations where the number of operations executed on objects is considerable.

References

1. Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. Legion: Enriching Internet Services with Peer-to-Peer Interactions. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 283–292, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.
2. Wyatt Lloyd, Michael Freedman, Michael Kaminsky, and David Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. pages 401–416, 10 2011.
3. Carlos Baquero and Nuno Preguiça. Why logical clocks are easy. *Communications of the ACM*, 59(4):43–47, 4 2016. Sem PDF. We would like to thank Rodrigo Rodrigues, Marc Shapiro, Russell Brown, Sean Cribbs, and Justin Sheehy for their feedback. This work was partially supported by EU FP7 SyncFree project (609551) and FCT/MCT projects UID/CEC/04516/2013 and UID/EEA/50014/2013.
4. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011.
5. Nuno Santos, Luís Veiga, and Paulo Ferreira. Vector-field consistency for ad-hoc gaming. In *Proceedings of the 8th ACM/IFIP/USENIX International Conference on Middleware*, MIDDLEWARE2007, page 80–100, Berlin, Heidelberg, 2007. Springer-Verlag.
6. Foursquare Labs, Inc. Foursquare. https://foursquare.com.