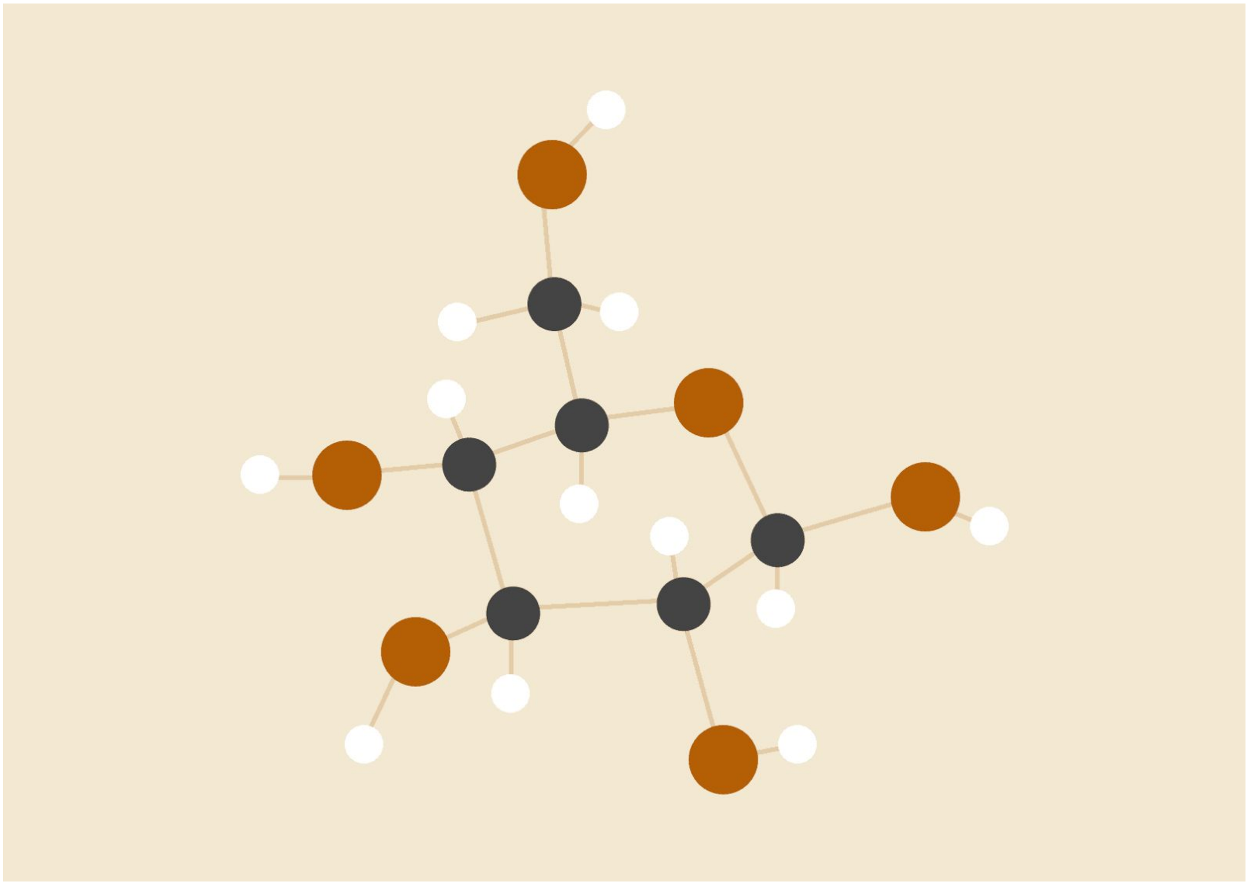
BLOB TAG

p2p multiplayer gaming system



Ameya Kamat, Yilan Xu

05.07.2017

TABLE OF CONTENTS

TABLE OF CONTENTS	**1**
INTRODUCTION	**2**
MOTIVATION	**2**
DESIGN	**3**
Zones	3
Game State Model	5
DHT IMPLEMENTATION	**7**
Inter-Node Communication	8
BOOTSTRAPPING	**10**
UI	**11**
Local Server	11
SUBSYSTEM INTERACTION	**12**
DISTRIBUTED SYSTEM CHALLENGES	**15**
Configuration	15
Consensus	16
Consistency	16
Fault Tolerance	17
DEMOS	**17**
Interim Demo	17
Final Demo	18
EVALUATION AND FUTURE WORKS	**18**
Node Failure	18
Network Failure	18
TEAM MANAGEMENT	**19**
Timeline	19
Team Management	20
REFERENCES	**22**

INTRODUCTION

Agar.io is an extremely popular multiplayer online action game released by Miniclip in 2015. The central idea of the game is allowing each user to control a circular blob, moving it in any given direction across a large 2 dimensional board. By consuming foods pellets, the blob enlarges and gains leverage in the game. When two players collide with each other, the smaller player is swallowed by the larger player.

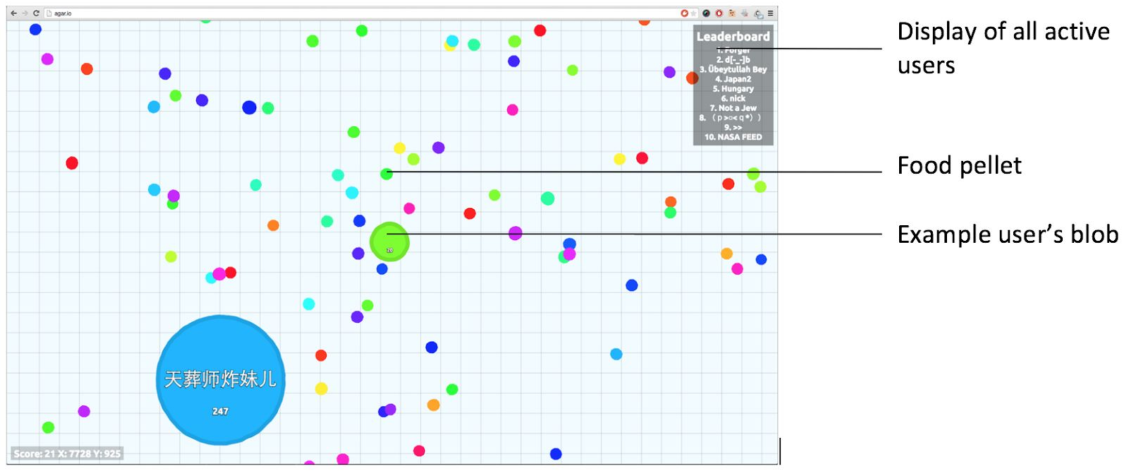


Figure 1.0: Agar.io UI

Apart from the simple addictive gameplay and graphics, one of the most attractive features of this game is that all the users play on a single map. The map is so huge, that only a fraction of the area fits on any given user's screen. Our project involves redesigning this game as a peer-to-peer game on a desktop client as opposed to a web-application based design. Our re-created game is known as BlobTag.

MOTIVATION

The original design for Agar.io is a client-server centralized mode, and is likely to have the following challenges:

- Streaming every user’s state changes, storing them internally, and then propagating them to the other users that happen to lie in the user’s screen. Doing this for multiple 100s of users in real time (1000’s of updates every second) is extremely difficult for a server, and likely requires several machines to maintain the game’ responsiveness
- The central server is a single point of failure for the game, and could bring down the system for all 100s of users if it crashes

Due to these reasons, we proposed and implemented a decentralized design for Agar.io that would mitigate these problems at a core level, and also offer the following benefits:

- Low cost of running the game: No game servers needed to keep the game running and responsive. Only a bootstrap server is needed to service initial requests on player joining, after which all game-related communication is handled directly with other players
- No single point of failure: Any node crashing or leaving the game will only affect the player represented by that node, and the game can continue normally for every other player.
- Scalability: A distributed design for Agar is highly scalable with minimal increases in cost. As for the centralized version, the cost to maintain the game increases exponentially for a large user base.

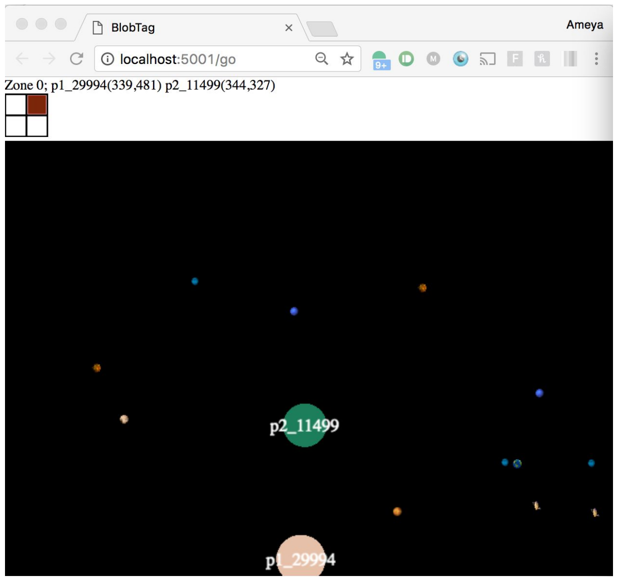
The following section outlines the design strategy we used.

DESIGN

Zones

In the original Agar.io game, the field of view of a player changes continuously as the player moves across the game board. In our implementation, the view changes are discrete. The game board is divided into a grid, where each block in the grid is known as a zone. The zone represents the view of any player’s screen. Each player can only have one active zone at any given time, and can only view other players in its zone. A player can move to a different zone by moving the sprite toward the edge of the zone. Once the sprite crosses over, it switches zone.

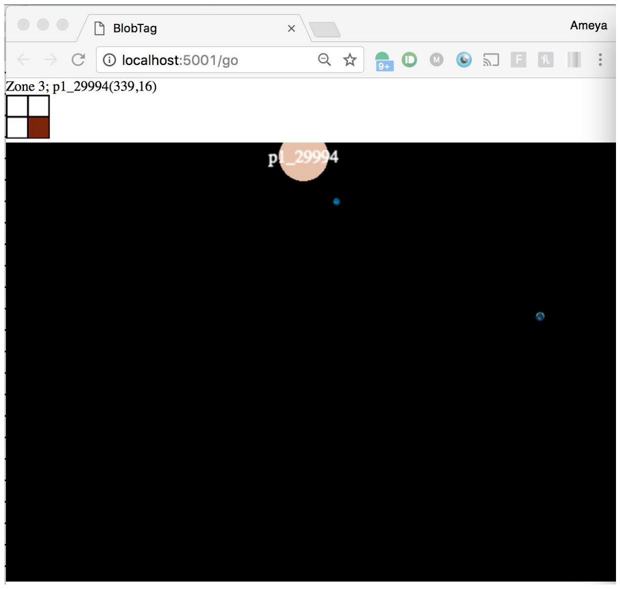
The following Figure 1(a) is a still taken from the game running on a 2x2 grid, from the perspective of *p1_29994*. The current zone is top-right (Zone 0), and the only 2 players in this zone are *p1_29994* and *p2_11499*. We can see *p1_29994* has approached the bottom boundary of the zone, and is about to change zones to Zone 3 (bottom-right).



Zone 0; p1_29994(339,481) p2_11499(344,327)

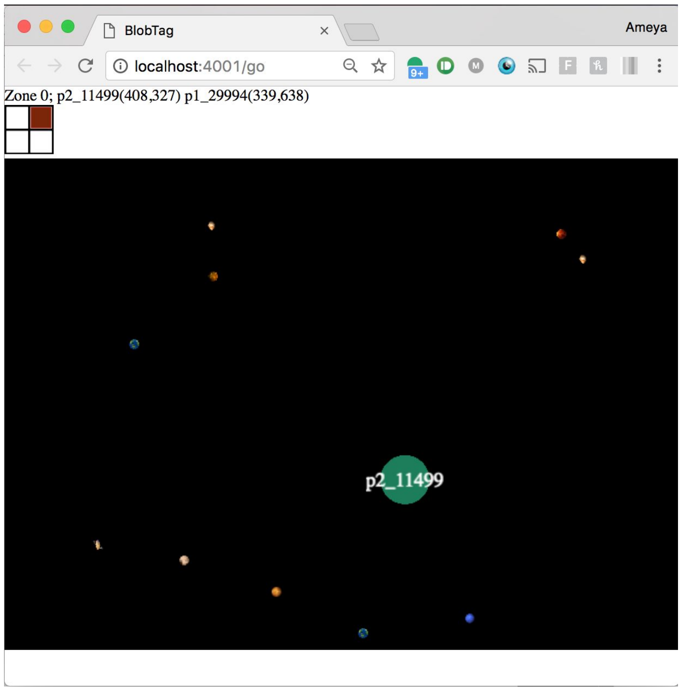
Figure 1.1(a): Still from BlobTag game before zone change: p1's perspective

As *p1* continues to move down, the game automatically changes p1's zone to Zone 3, as depicted in the following figures. Zone 3 happens to have no other players.



Zone 3; p1_29994(339,16)

Figure 1.1(b): Still from BlobTag after zone change: p1's perspective



Zone 0; p2_11499(408,327) p1_29994(339,638)

Figure 1.1(c): Still from BlobTag after zone change: p2's perspective.

Game State Model

As we try to mimic the Agar.io gameplay as closely as possible, we decided to encapsulate a player's (blob) state in these fields (<name> <type>) which would allow any modern GUI to accurately render a sprite for the player:

```
BlobInfo struct {

	PlayerName  string

	IsAlive     bool

	Color       int

	Xsize       float64

	Ysize       float64

	Xvel        float64

	Yvel        float64
```

```
        Xpos        float64
        Ypos        float64
        ZoneId      int
        ZoneChanged bool
}
```

As for the state of the entire game, it only consists of the following items:

- A list of all active players (blobs) in the game, and their respective states
- A list of all food pellets and their locations

As there are a large number of pellets (around 5x the number of players in the original game) and are not important in the gameplay apart from fortifying players' blobs, we decided to leave them out of the game state for our implementation. After all, we don't have any guarantee that the original game maintains consistent food pellet positions across all players' screens since they are just too many in number.

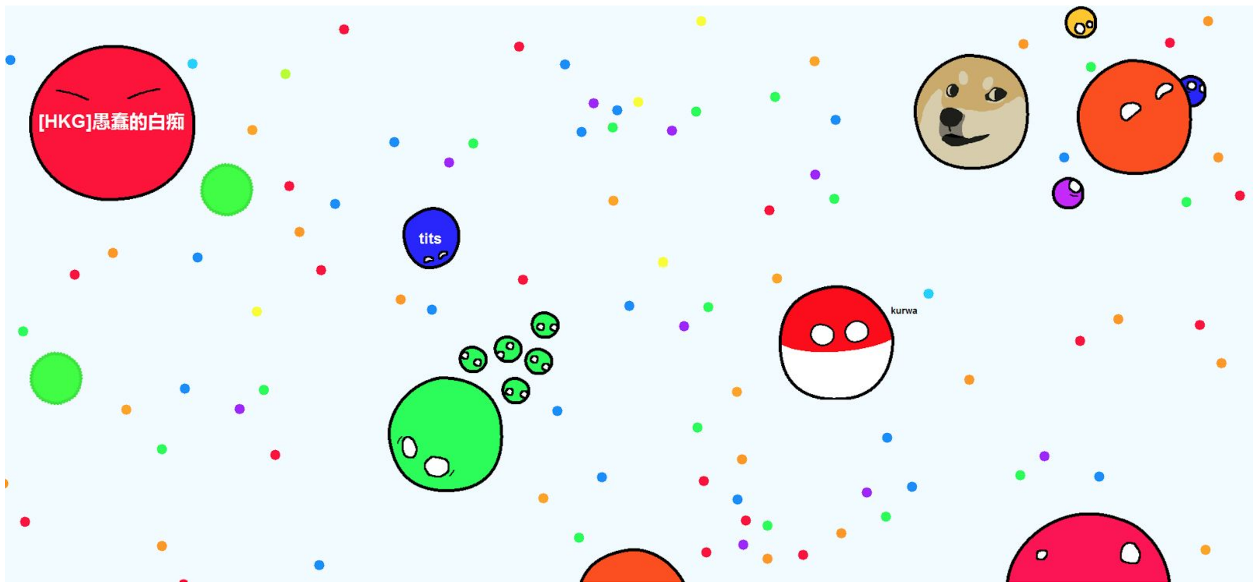


Figure 2.0: Agar IO screen example: Food pellets are over 5x the number of player sprites

As in our system there is no centralized mechanism to maintain game state and propagate updates to players, the game state has to be maintained on players' devices. Player information is queried / multicast from / to nodes that need them. For this purpose, our game this uses a Distributed Hash Table (DHT) to store chunks of the game state in each of the players' devices. No single device knows about the entire game board, but the information on all the devices collectively encompasses the board state. This naturally implies that the game is packaged and distributed as a native desktop application using network connectivity as opposed to a website-based game.

One crucial aspect about using a DHT in this context is that the pattern of key access is predictable, and can be leveraged in the design. As mentioned in the previous section, each player only views a small portion of the game board, and only interacts with other players within his/her zone. A player can transfer to a different zone, and interact with players in the new zone, but there can be only one current zone at a time. This brings us to the optimization that each node in the DHT only needs the most up-to-date information from other players in its own zone. As for players in other zones in the game, they are invisible to the player, so it is sufficient to just know their IP addresses, but not necessarily about their exact positions, velocity, size, etc.

Thus, in summary, the decentralized design of BlobTag involves creating a Peer-to-peer network of nodes in the game, using a DHT to distribute and propagate game state across nodes, such that only nodes in the same zone receive updates in real time.

DHT IMPLEMENTATION

Due to its modern concurrency and networking primitives, and also its unique garbage collection style, we decided to implement the DHT in Go. While there are several Go packages that implemented distributed hash tables such as Wendy, we quickly realized that they do not offer much flexibility with routing and data storing. Since our implementation needs a specialized routing and key distribution system that dynamically changes during game play, our only option was to implement the DHT from scratch.

The following structure was used to represent a node in the DHT:

```
type DhtNode struct {

     NodeId   int
```

```
        IpAddr   net.IP

        Port     int

        BlobInfo *BlobInfo

        //time of last updating this struct (epoch time)

        UpdateTime int64

}
```

The node contains an IP address on which it listens for queries, as well as the information about the blob's current state. The NodeId is a unique identifier for each node assigned at startup. It is used as the key in the DHT, the value being the DhtNode object.

Inter-Node Communication

RPCs are used to communicate among nodes via a pre-declared set of queries. This is an optimal choice for communication for a number of reasons:

- RPCs have a natural query-response system, so there is no overhead of sorting incoming messages and matching them with the correct queries
- RPCs are well suited to sending structured information over the internet (such as nested structs containing node and game information), once the marshaling and unmarshaling operations have been defined
- Golang has well documented support for its RPC package

The structure of an RPC function declaration in Golang is of the format:

```
MethodName(argType *T1, replyType *T2) error
```

The sender creates a query struct (argType), and calls the RPC over a HTTP connection to a remote server. The golang package marshals the query struct and sends it to the server. On receiving a reply from the server, the package unmarshals the response and populates the replyType struct, which can then be accessed by the user like a local object.

In our application, each node plays the role of server and client. Each node must listen for queries and updates regarding data it holds (server role), as well as send updates to other nodes' servers on change in game state (client role). The RPC server listens at all times (since the application is opened) for queries on a port specified by the user. The DHT exports the following RPC interface:

- `Get(NodeId *int, reply *DhtNode)`
    - This is a simple query for node information based on node ID.
- `Update(updateNode *DhtNode, success *int)`
    - This is a request signalling a node that:
        - If it does not contain an entry for updateNode, then add updateNode to its table
        - If it already contains an entry, then update it with the value specified in updateNode
- `BroadcastUpdate(updateNode *DhtNode, reply *List<DhtNode>)`
    - This is similar to an Update request, but in addition, the update is forwarded to all the nodes in the system.
    - Each set of BroadcastUpdate message is tagged with a unique hash value generated by the origin node, to make sure any duplicates received from multiple paths in the network are dropped
    - Note that the reply type is a list of all nodes. This is useful for new nodes joining the network- when they send a BroadcastUpdate to one of the nodes, not only is their information propagated to other nodes in the network, but they also receive a list of up-to-date node objects that they can use to contact nodes in their zone.
    - Note that since BroadcastUpdate messages are flooding type, they are used minimally during the game. The only 2 use cases are the following:
        - New node joins the network
            - This is done to inform all players about the new node, so players in the same zone as new node can update their UI to reflect the new player.
        - Node changes zone
            - Ideally, when node N moves into zone Z, it could do a ZoneUpdate (see below) to nodes in zone Z to update them. However this is insufficient as there could have been nodes that entered Z that N does not know about. This would imply only a partial update to nodes in Zone Z. Thus it is important to Broadcast this zone change to inform all active nodes about this change.
- `ZoneUpdate(updateNode *DhtNode, reply *List<DhtNode>)`
    - This is similar to BroadcastUpdate, except that the forwarding is only done

    to nodes in the same zone as the original sender
    - This function is called most often during the game

While this interface provides for updating the DHT during the game, the joining process is an independent routine. This routine is discussed in the following section.

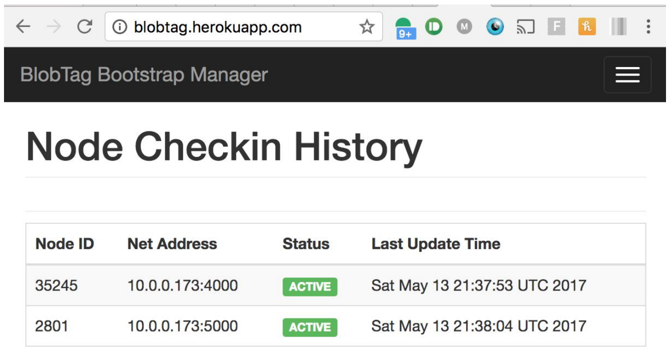
BOOTSTRAPPING

When a user launches the application, it needs a mechanism to locate other nodes in the network in order to join the DHT (and game). For this purpose, there is a remote bootstrap server that informs any new node about all the nodes in the network, along with their internet addresses.

The bootstrap server is an independent web application deployed to the Heroku cloud, accessible at blobtag.herokuapp.com. This bootstrap web application is also written in Go, and offers the following features:

- Receive updates from new nodes
    - New nodes joining the network make an update HTTP POST request to the server. The server logs its IP address and port
    - These logs are stored in persistent storage (CSV file), so that the bootstrap server can restore state after a failure
- Get current nodes in the network
    - The server handles a 'get-all' request, responding with a JSON representation of all the nodes and their addresses from its database. This information allows new nodes to bootstrap by connecting to one or more of these nodes and joining the DHT
    - Note that since nodes can arbitrarily leave the network or crash, not all the nodes in the returned list are guaranteed to be active. It is the responsibility of the nodes to handle this by only updating their table with nodes that are active

On receiving the JSON object containing the list of all nodes in the network, the new node can then contact them one by one to (1) determine if they are active, and (2) get their updated information. Then, the new node can send a ZoneUpdate to update the players in its zone about the new player.



Node Checkin History

Node ID	Net Address	Status	Last Update Time
35245	10.0.0.173:4000	ACTIVE	Sat May 13 21:37:53 UTC 2017
2801	10.0.0.173:5000	ACTIVE	Sat May 13 21:38:04 UTC 2017

Figure 3: Bootstrap server dashboard page depicts nodes that it was contacted by

UI

As Go is fairly limited in terms of graphics packages, we decided to develop a web-based UI that can leverage powerful Javascript / WebGL -based rendering for the game. Additionally, having a web-based UI would also make the game platform independent, as it would be ultimately efficient to have a single package that runs on the local browser.

Our implementation uses PixiJS, a WebGL-based game engine that runs in an HTML canvas, capable of drawing and moving sprites, detecting collisions, etc. This introduces a new component to the BlobTag implementation: the local server.

Local Server

The local server is written in Golang with the primary objective of connecting the web UI with the DHT. When the Blobtag application is run, a web browser is opened and pointed directly to the port that this application runs on. This server runs concurrently with the DHT RPC server. In order to pass on game state changes from the UI to the DHT and vice versa, the following communication mechanisms are used:

- Websocket server
    - The Blobtag web server runs with a websocket server alongside it. On loading the web page, the client-side Javascript program connects to this server and subscribes for UI updates.
- AJAX requests
    - The Javascript UI listens for button presses from the user. Any of the arrow

    - button presses change blob state. These changes are passed onto the websocket via AJAX requests.
- Go channels
    - As the web server runs in parallel with the DHT RPC server, Go channels serve as a mechanism for inter-process communication between these 2 threads. For example, when updates from a node are received by the RPC server (eg. ZoneUpdate calls), the received objects are passed from the RPC server thread to the web server thread over a Go channel. The object is then marshalled by the web server into a JSON and sent over the Websocket channel to the client side UI.

The above communication mechanisms use JSON representations that can be used to directly create PIXI.js sprites or `DhtNode` objects. This allows for direct state communication between the UI and DHT.

The following figure illustrates these communication channels:

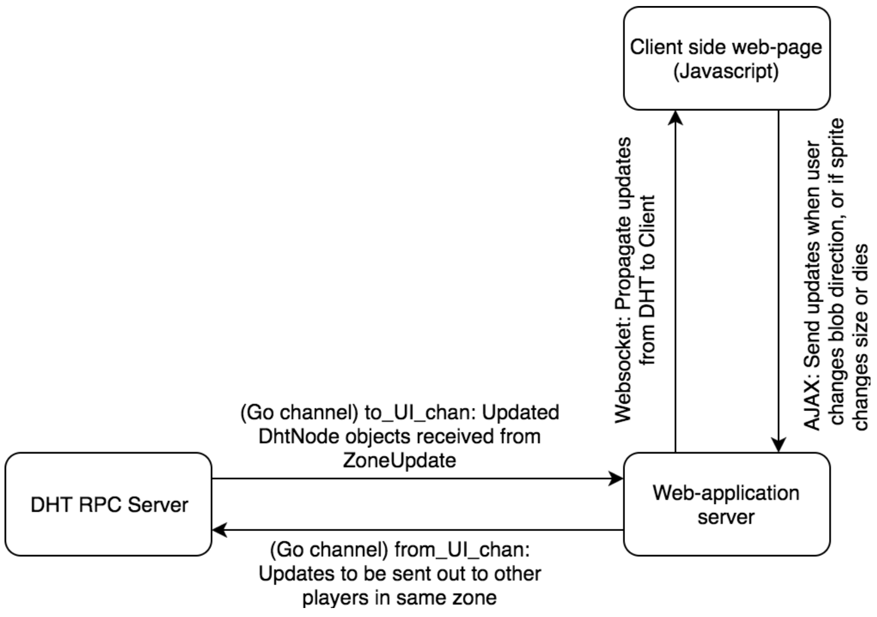

Client side web-page (Javascript)
Websocket: Propagate updates from DHT to Client
AJAX: Send updates when user changes blob direction, or if sprite changes size or dies
(Go channel) to_UI_chan: Updated DhtNode objects received from ZoneUpdate
DHT RPC Server
Web-application server
(Go channel) from_UI_chan: Updates to be sent out to other players in same zone

Figure 4: Illustrating the interaction of the 3 subcomponents of the BlobTag application

SUBSYSTEM INTERACTION

Since there are several components to this application running concurrently, an interaction diagram with an example use case would be a good tool to grasp the working of the system as a whole. The following interaction diagram illustrates the sequence of events when User (a person) launches the Blobtag application. The game has only 2 other players (for simplicity), and one of them is in the same zone as User.

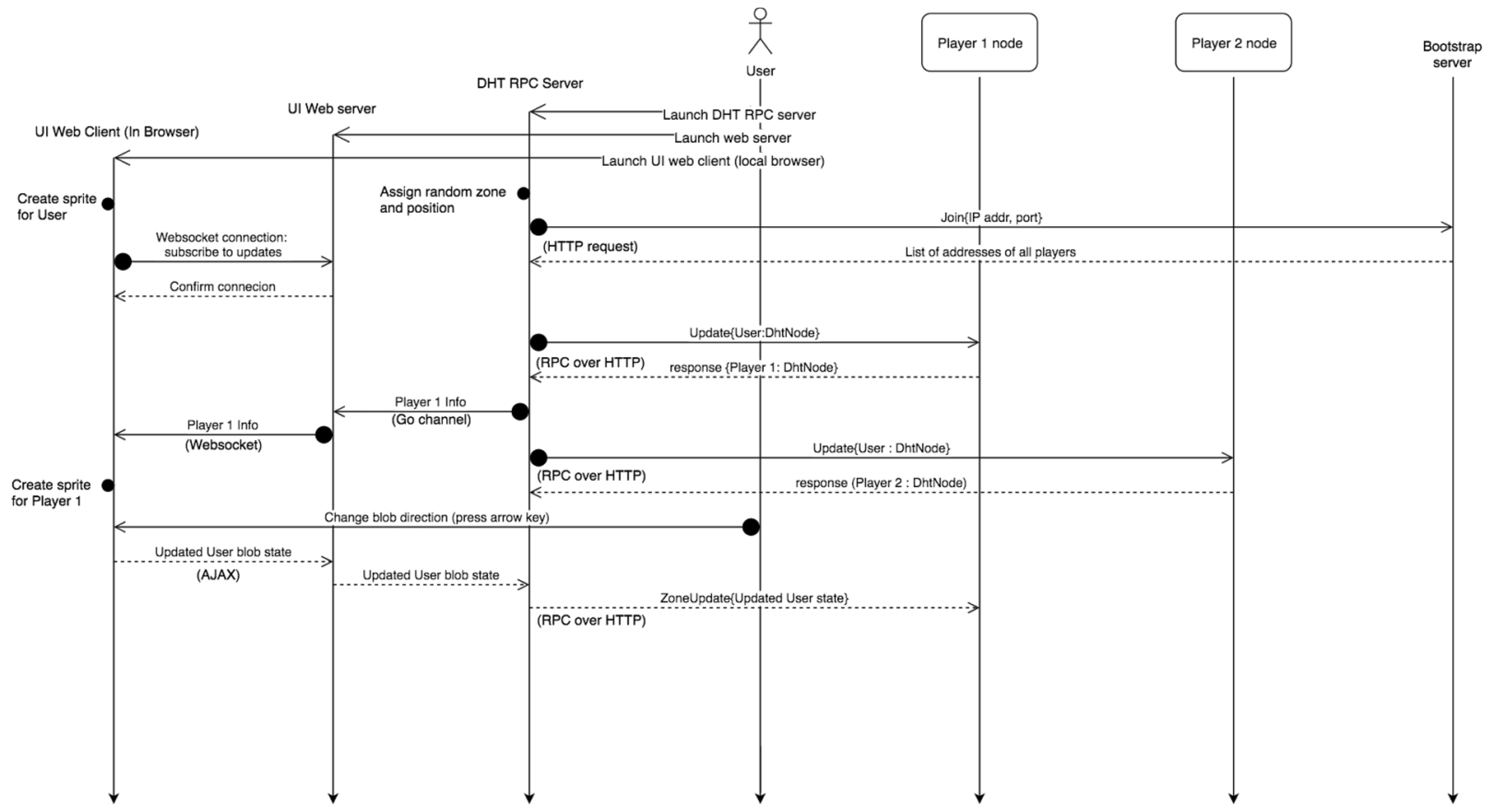


Figure 5: Subsystem interaction example

DISTRIBUTED SYSTEM CHALLENGES

To examine the BlobTag project in terms of the distributed systems challenges, we found that each of the 4 areas were explored in a different way:

Configuration

By implementing a distributed hash table from scratch, defining a shared game state, implementing the inter-node communication layer using RPCs, and implementing a bootstrap server application, there were several node configuration challenges to overcome. The following figure depicts the configuration mechanism for a single node:

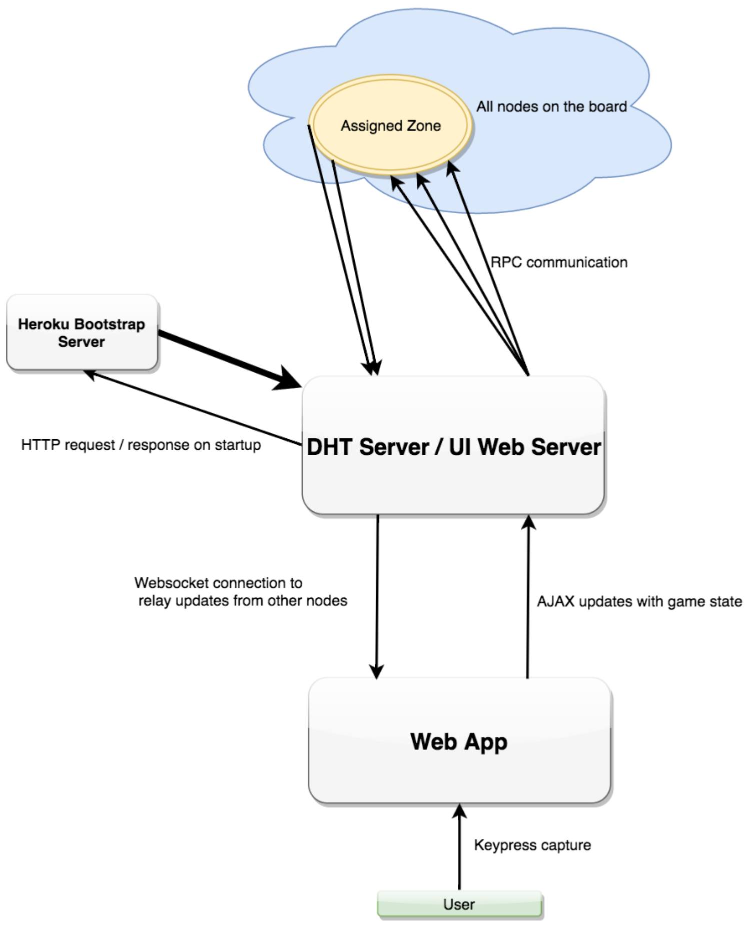

Assigned Zone
All nodes on the board
RPC communication
Heroku Bootstrap Server
HTTP request / response on startup
DHT Server / UI Web Server
Websocket connection to relay updates from other nodes
AJAX updates with game state
Web App
Keypress capture
User

Figure 6: Illustrating the configuration mechanisms for a node

Consensus

As there are can be several nodes in a single zone, it is likely that conflicting update messages are sent from different sets of nodes. As traditional consensus algorithms such as Paxos require a 4-stage process involving a majority of nodes (at least order O(N)), we predicted that the delay that this would cause would not be acceptable for this kind of live action game. This led us to design rules that would make it easy for nodes to resolve a situation where conflicting information is received from different nodes:

- The Mind Your Own Business rule.
    - Each node is only allowed to propagate updates about its own state to the network. IE, consider the situation where players A, B, C, D and E are in the same zone. A notices that B collides with C and consumes it. A is not allowed to send a ZoneUpdate message regarding this event.
    - The only exception to this rule is for node consumption. If A consumes B, then A has the right to send an update about player B’s death.
- Non Resurrection rule
    - If an update is received that player X is dead (IE consumed), then no future update can bring player X back to life, even if the update specifically says so.

These rules allow nodes to resolve conflicts in updates without wasting time exchanging messages and causing significant lag for the user.

Consistency

As BlobTag is a real-time multiplayer action game, the design choices involving the representation of node state in the DHT (see BlobInfo struct) and zoning choices are crucial to maintaining a consistent state across players’ screens:

- Encapsulating position, velocity and time in the struct means that periodic updates are not necessary (as in Agar.io). Each player can estimate the other's’ positions based on the velocity and last time of update. Reducing the number of updates in a dense network serves to reduce the network delay of each message, leading to a more consistent game state across nodes
- Creating discrete zones as opposed to a continuous map allows each node to use the same coordinate system for any given zone, making it easier to maintain a consistent view of the board

- In-zone flooding
    - This is the algorithm that is invoked on calling ZoneUpdate: on receiving a ZoneUpdate, a node not only updates its table, but also forwards the message to other same-zone nodes not in the seen list. Before forwarding a message, a node adds its own ID to the seen list. This mechanism ensures consistency when there are message drops or network partitions, as there are multiple routes to any recipient

Fault Tolerance

When a node leaves the game, the system will handle the game termination gracefully, as it just behaves as if no updates were received. So it will moves under the last recorded setting (speed, direction and size) until it reaches the end of board and moves out or consumed by one of the active blob on the board.

On restart, the node contacts the bootstrap server to receive IP addresses of active nodes. Since the bootstrap server stores state in a CSV file, the node can restart with previous setting after failure.

If one of the node in the zone terminates unexpectedly, there are potentially two situations: First, the node is the only node in its zone, then the leaving of the node won't affect any other nodes on the board as nodes active in other tiles don't have to contact with the node anyway. The second situation is there are more than one node in the zone, and in this case, since the nodes in one zone multicast update message to all members in the zone, the active nodes can still be able to update itself with the external update messages.

CHALLENGES

The following challenges and learnings were encountered during course of working on this project:

- Learning Go: memory management, concurrency primitives (Goroutines and

channels)
- Working with Go's synchronous RPC API
- Web application
    - Working with Go's HTTP web server; implementing request handlers for different functions
    - HTML templating
    - Using an external Websocket library in go
    - AJAX
    - Game development using PIXI JS: Working with the HTML Canvas, running an animation loop, sprites, collision detection, etc
- Heroku
    - Creating runtime container for Go web application
    - Using Go-vendor to specify dependencies
    - Specifying on-launch commands via a Procfile and Dockerfile

DEMOS

Interim Demo

Demonstrated our system works with broadcasting mechanism, each node stores its own information and push update messages to the network by broadcasting the information to all nodes known to the user. However, the users have to have knowledge of each other's' IP address and port number to start a connection and see the updates from the other end. Also, another issue was the update messages are multicast to all users in the game, thus it is more easily to have networking issue and waste bandwidth. So the game didn't scale well.

Final Demo

Created a binary distribution for OSX (available here), with the application running with full functionality. Demonstrated multiple nodes in single zone, propagation of game state in node (changing direction, consuming food pellets, consuming other blobs).

EVALUATION AND FUTURE WORKS

Some of the failure scenarios we considered are:

Node Failure

Unexpected node leaving: if one of the users leaves the game improperly, there will be no time to transfer the data hold by this user gracefully, and it might cause jitters or even lost of current game states if we cannot recover the data from the back up efficiently.

This failure scenario is discussed in fault tolerance section, we uses the last known setting to render the node on the screen until it moves out. Even we lost one of the node in the zone, we still have alternative path to ask for updates due to the implementation of DHT, so the node failure won't be a problem in our system.

Network Failure

If the network situation of certain area becomes unstable, the game, just like any other web-based application won't work for users. There is nothing we can do to prevent the issue or recover from the problem. However, since our game group is formed based on the network location (IP address), the impact of network failure in certain area should be limited to those groups.

Logger Failure: The logging information is stored in a centralized DNS server, so the possibility it goes down is tiny. However, we still consider this as a possible fault in our system, and we may also consider implementing a replica for the logger.

In our implementation, each node maintain the list of the neighbors it abuts, so even the bootstrap server fails, the node can still be able to retrieve updates. However, new user may have trouble entering the game, but users already in the game won't be affected, and even be able to restart the game after node failure.

We tested our design with at least 8 users play simultaneously and resulted no faulty scenarios. Comparing with the centralized game, Agar.io

- We have better scalability. Using dht we automatically distribute nodes in different section of the board, so a node only need to maintain connections with a subset of the nodes. The game can be scaled by increasing the board partitions.
- No single point failure and communication bottleneck. There is no centralized server holding all the information, we don't need to worry about the server go

done and all nodes lost its current state.
- Its ad-free!

To improve on what we have now, we propose to add more game features like mouse-control, node splitting and accelerating. Also, we still have some overheads in the system, a better DHT algorithm using spanning tree routing might reduce the redundancy in the system.

TEAM MANAGEMENT

Timeline

Date	Task	Note
Mar 27	Project Design Document	Project Implementation Proposal. Give an overview of the project, and think about how to design the project.
Apr 11 (week 12)	Interim Demo	Demonstrate some level of functionality of the project or some implemented subtasks. Evaluate the progress with the mentor and professor.
May 9 (week 16)	Final Project Demo	Demonstrate our final result meet the goal we made in the proposal.
	Final Project Report	Submit a detailed report on the implementation of the project and evaluation of the final result.

Based on the dates above, we set several milestones:

	Date	Task
Milestone 1	Mar 24	Project Feedback Review
Milestone 2	Mar 31	Finalize project implementation plan after meeting with mentor/professor
Milestone 3	Apr 8	Interim Demo Testing
Milestone 4	Apr 16	Review the project implementation with mentor/professor
Milestone 5	May 2	Subtasks Testing
Milestone 6	May 4	Final Project Testing

Team Management

We identify several subtasks of the project:

	Subtask	Assigned
Project Selection	Brainstorm	Ameya & Yilan & (Astha)
	Idea Evaluation	
	Project Proposal Video	
UI/GUI	Implement basic game functionalities	Ameya
	Add graphical features	
	Identify relevant actions	
	Construct message type to	

	exchange data	
Distributed Middleware	Distributed keyspace	Yilan
	Implement CAN-like multicast functionality (Message Routing)	
	Routing Mechanism Testing	
Network Infrastructure	Bootstrapping	Ameya
	Node joining	
	Node leaving	
Testing	Subtasks testing	Ameya & Yilan
	Performance testing	
	Fault tolerance testing	
Demo	Interim Demo	Ameya & Yilan
	Final Demo	
	Final Report	

REFERENCES

1. Agar.io Website: http://agar.io/
2. Etherium: https://github.com/ethereum/go-ethereum/wiki/Peer-to-Peer
3. TenderMint: https://github.com/tendermint/go-p2p
4. Content Addressable Network:

https://people.eecs.berkeley.edu/~sylvia/papers/cans.pdf

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.365.9178&rep=rep1&type=pdf

http://www.utdallas.edu/~kxs028100/Papers/CAN.pdf

5. Building a Peer-to-Peer Multiplayer Networked Game

https://gamedevelopment.tutsplus.com/tutorials/building-a-peer-to-peer-multiplayer-networked-game--gamedev-10074