

Foodchain

An interactive ecosystem

Nancy Qian

Objective:

Explore and *simulate* environmental scenarios

How do different variables affect a Foodchain?

What is the cause of Foodchain imbalance?

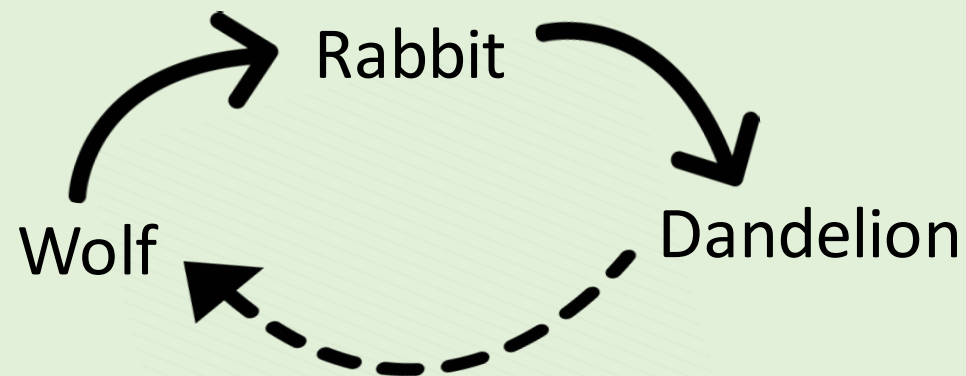
Which organism has the highest survival rate?

What and is there an optimal Foodchain?

What is in the Foodchain environment (initially)?

<ul style="list-style-type: none">• Animals<ul style="list-style-type: none">- Wolf- Rabbit	<ul style="list-style-type: none">• Plants<ul style="list-style-type: none">- Dandelion
--	---

Foodchain hierarchy (Predator -> Prey)



What are the *variables* in the ecosystem?

Weather condition (Rainfall): factor multiplied to plant variables in order to affect growth, birth, etc.

- Intensity (Range: 1-10)

Birth periods: periods of time between reproduction counted by **ticks***

- Wolf born period (Range: 1-400)
- Rabbit born period (Range: 1-400)

Birth rate: the percentage rate of successful reproduction every birth period

- Wolf born rate (Range: 1-100%)
- Rabbit born rate (Range: 1-100%)
- Dandelion born rate (Range: 1-100%)

Exploration:

Scenarios with differing Independent
Variables

Scenario #1: Light Rainfall (weather condition)

Variables

- Weather condition= Lowest (1)
 - Birth periods= Initial (200)
 - Birth rates= Initial (50%)

Description:

A situation in which a habitat experiences abnormally low levels of rainfall.

Results (Data):

- Video Data

Timestamps:

1. At 00:01:10, Dandelions die out
2. At 00:01:58, Rabbits die out
3. Lastly, at 00:02:10, Wolves die out



Conclusion:

- The dandelion died out first due to the lack of rain (born period impacted)
- This then caused the rabbits to die out from lack of food source, dandelion
- Lastly, the wolves die due to lack of rabbits
- Shortage of rainfall causes **imbalance** of the Foodchain and is not optimal

Real World Application

- An example of this could be the deserts or droughts
- Lack of rainfall or bodies of water
- **This is a problem because they:**
 - Plant growth depends on rainfall
 - Land becomes infertile
 - Does not support life (lack of water)

Scenario #2: Shortened Wolf born period

Variables:

- Weather condition= Initial (5)
- Wolf born period= Lowest (1)
- Rabbit born period= Initial (200)
 - Birth rates= Initial (50%)

Description:

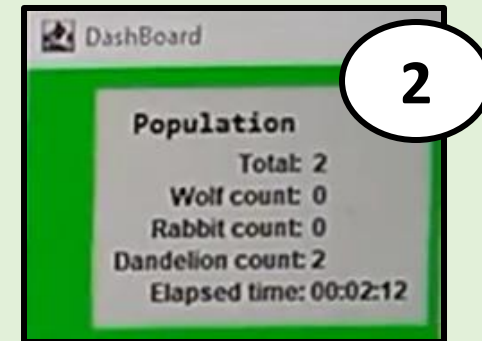
A situation in which the habitat's wolves reproduce at an abnormally high rate; period of time between each birth shortened

Results (Data):

- Video Data

Timestamps:

1. At 00:01:06, Rabbits die out
2. At 00:01:36, Wolves die out
3. Dandelions keep growing (may eventually die out)

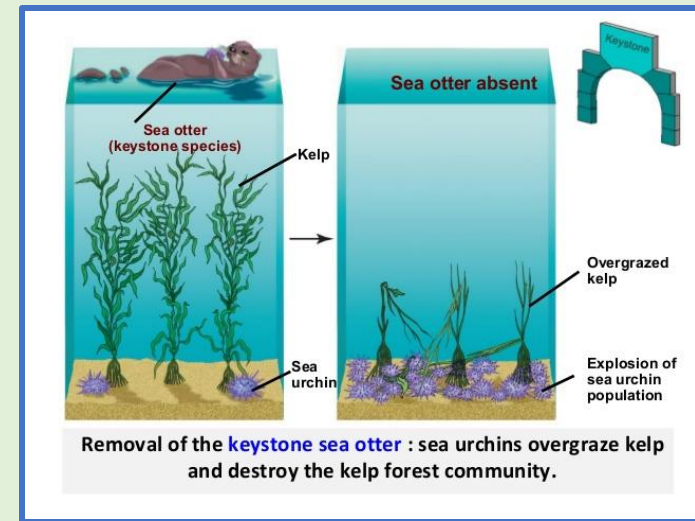


Conclusion:

- The rabbits died out first due to the amount of wolves (produced by the low born period)
- This then caused the wolves to die out from lack of food source, rabbits
- The dandelion aren't fatally affected by this since their predator rabbit died first
- A large production of wolves causes **imbalance** of the Foodchain and is not optimal

Real World Application

- An example of this could be keystone predators
- **This means:**
 - Predators introduced to moderate prey population and competition



- **This is could be a problem because they:**
 - Could hunt prey population to extinction

Scenario #3: Shortened Rabbit born period

Variables:

- Weather condition= Initial (5)
- Wolf born period= Initial (200)
- Rabbit born period= Lowest (1)
 - Birth rates= Initial (50%)

Description:

A situation in which the habitat's rabbits reproduce at an abnormally high rate; period of time between each birth shortened

Results (Data):

- Video Data

Timestamps:

1. At 00:01:06, Dandelions die out
2. At 00:01:36, Rabbits die out
3. Lastly, at 00:02:06, Wolves die out



Conclusion:

- The dandelion died out first due to the amount of rabbits (produced by the high born rate)
- This then caused the rabbits to die out from lack of food source, dandelion
- Lastly, the wolves die due to lack of rabbits
- A large production of rabbits causes **imbalance** of the Foodchain and is not optimal

Real World Application

- An example of this could be the [European rabbits in Australia](#)
- **European rabbits:**
 - Have a lack of natural predators
 - Can give birth to more than **four litters** a year with as many as **five kits** (baby rabbits) each (**high birth rate**).
- **This is a problem because they:**
 - Drive out native species from their homes
 - Compete for food and other resources
 - Loss of plant biodiversity

Exploration Takeaways

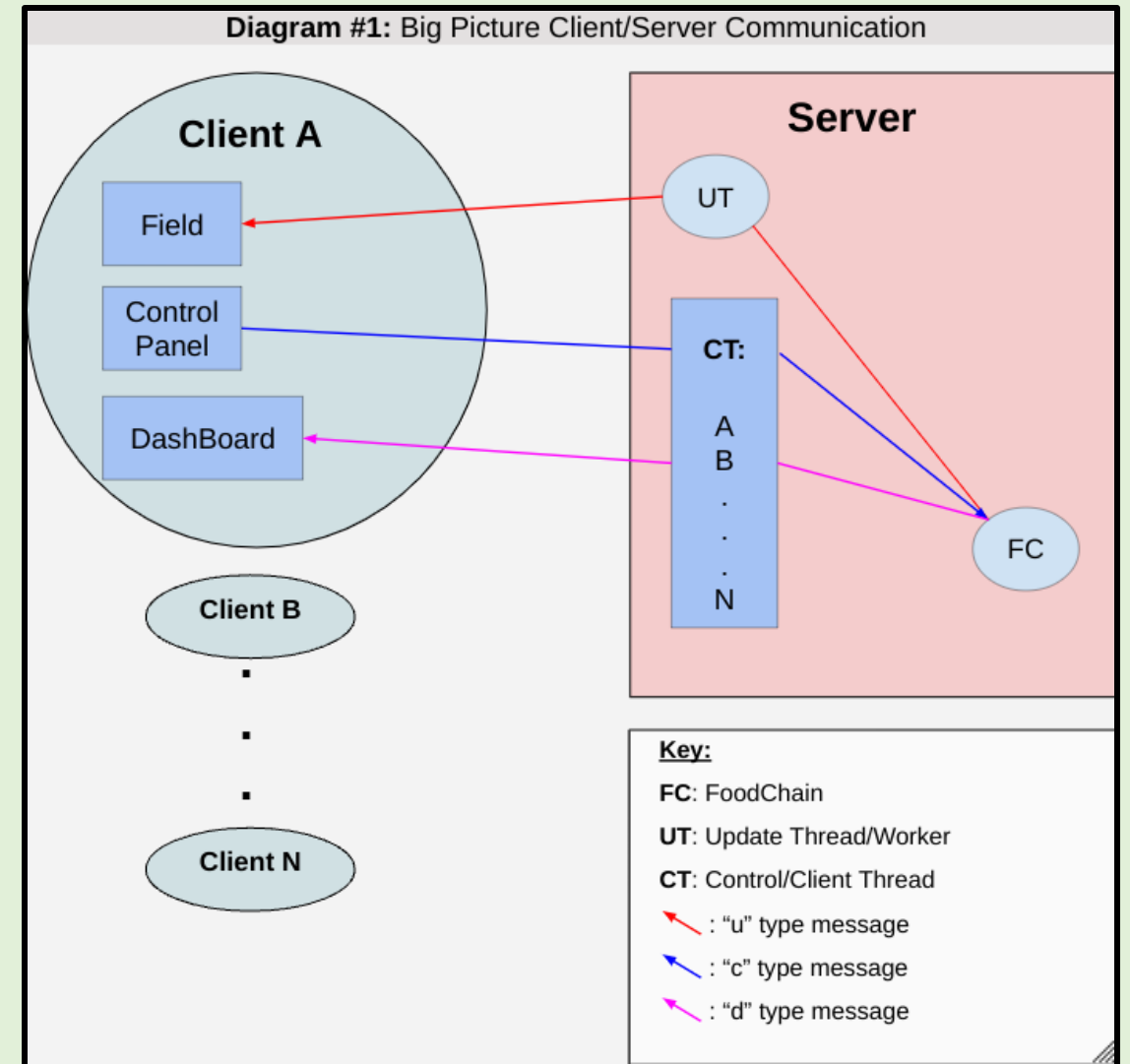
- Foodchain imbalance could be caused by:
 - Due to drastic changes in the variables, some part of the Foodchain is eliminated, the entire chain of predators before it that relied on it for food would be affected
 - Example from our Foodchain:
 1. Wolf->Rabbit (high birth rate)->~~Dandelion~~ (all eaten by Rabbit; extinct)
 2. Wolf->~~Rabbit~~ (high birth rate but now; no food source; all eaten by Wolf; extinct)
 3. ~~Wolf~~ (no food source, extinct)
- Introduction of **non-native species** with extremely high birth rates, such as the rabbit, could end up causing plant species, like the dandelion, **to go extinct**, which causes the rabbits to go extinct as well, if dandelions were a crucial part of their diet; this also impacts the wolves who hunt the rabbits for food (the WHOLE Foodchain affected)

Simulation:

Creating the Foodchain

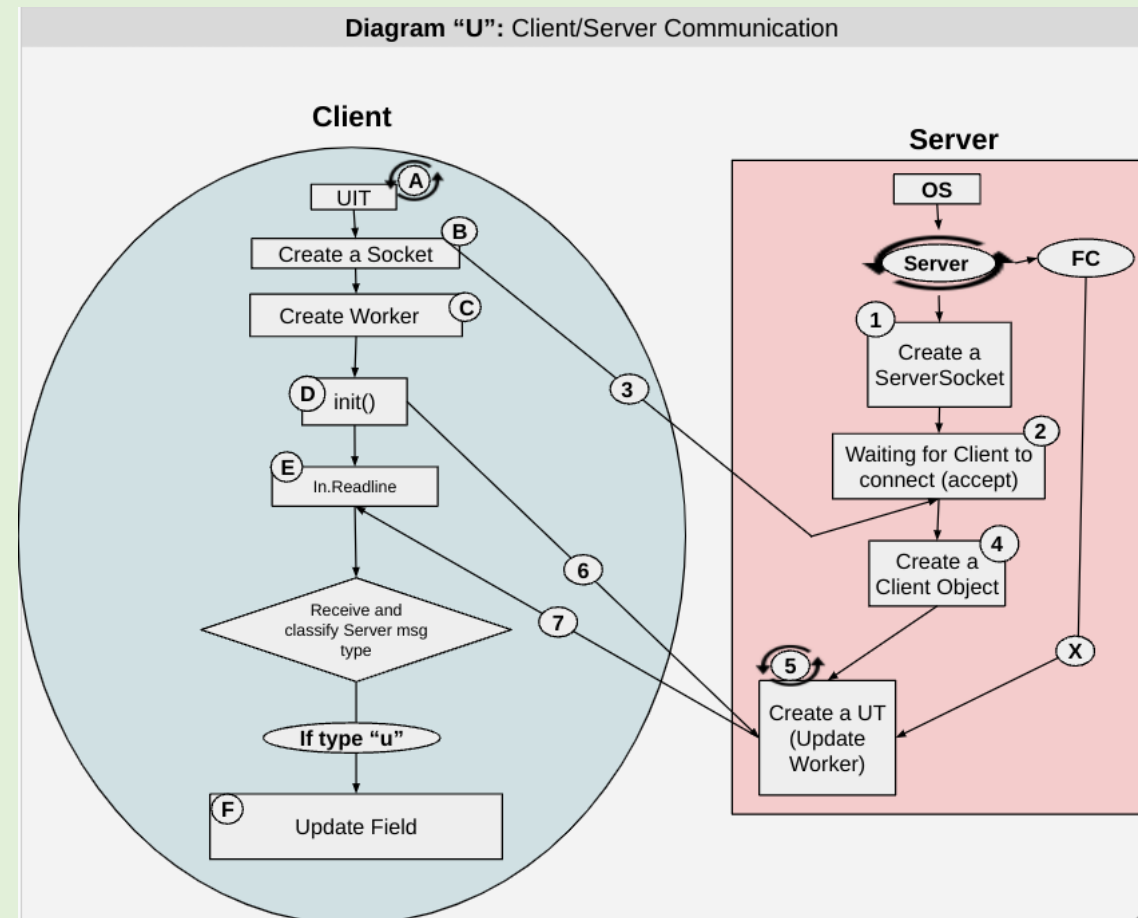
The Broader Picture

- Create a **Server** with a running environment
- Multiple **Clients** from different platforms that can connect to Server to view environment



"U" messages: Update Field UI

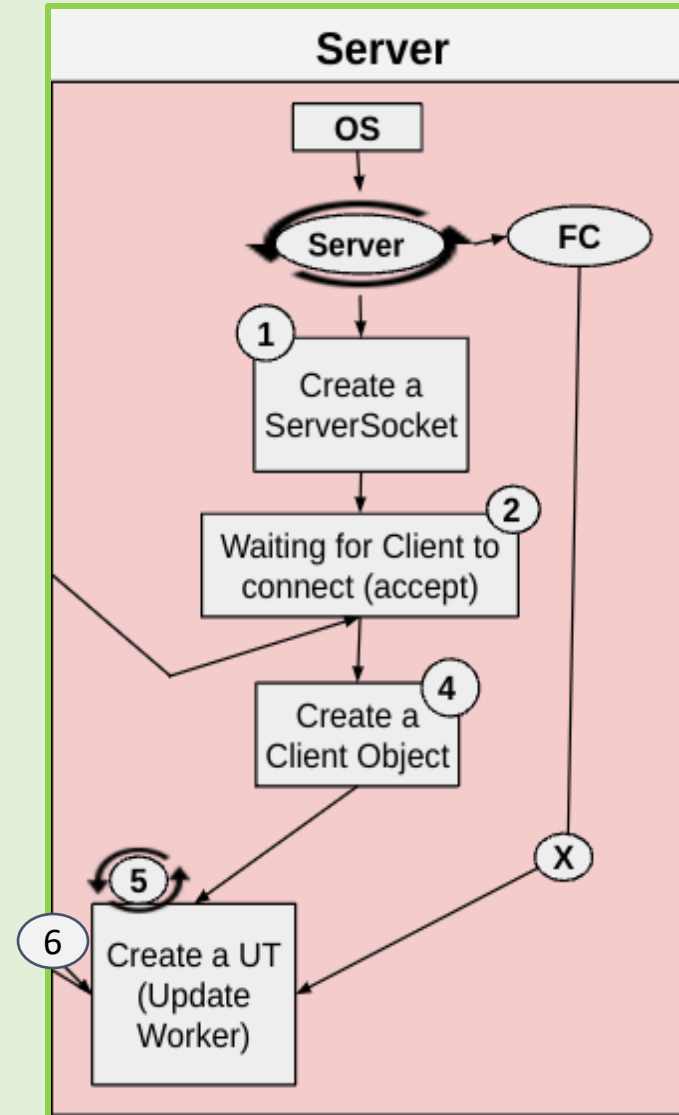
Server's Update Worker constantly pushes values to update FoodChainField (state of animals, position, environment)



"U" messages (Update Field UI): Server side

Tool Tips:

- 2. Call accept of ServerSocket (block Server control)
- 7. Constantly broadcast to all CT in Client List
- X. Update Worker constantly updates using FoodChain class's Life List (thread list)



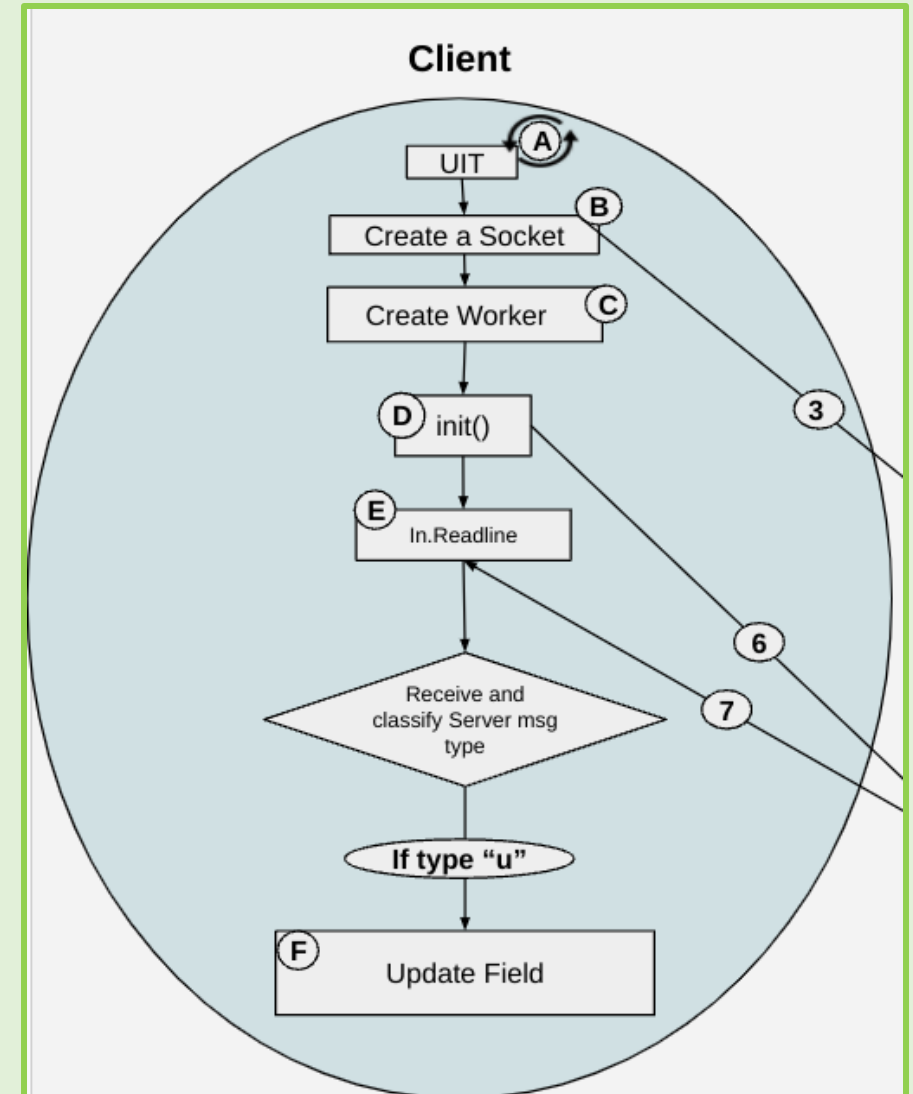
"U" messages (Update Field UI): Client side

Tool Tips:

A: create a UI Thread

B: create a Socket (unblocks Server that was waiting for Client response)

C: create a Worker Thread for Client



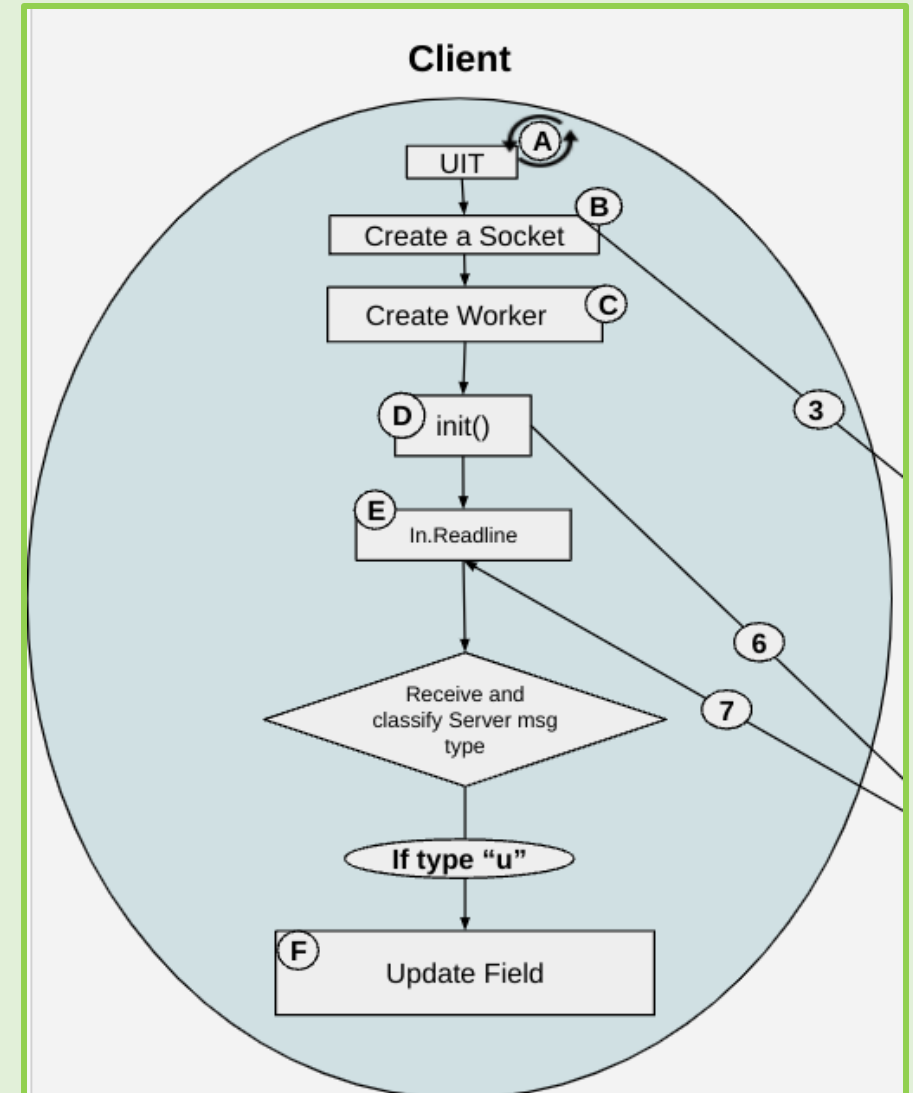
"U" messages (Update Field UI): Client side

Tool Tips:

D(6): call init() method to send (out) key from Client that need initial slider values

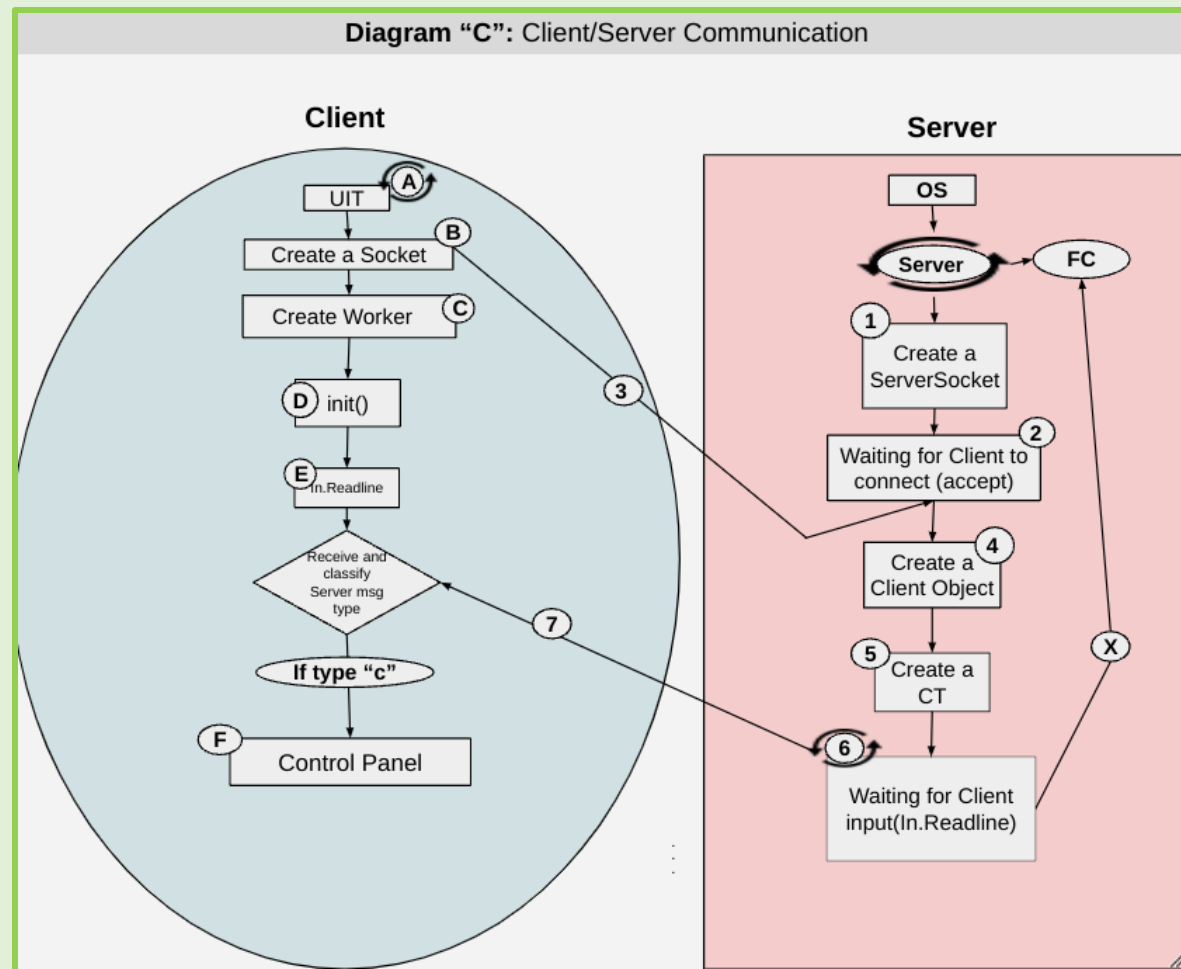
E: waiting for msg from Server (in)

F: Update FoodChainField UI



"C" messages: Update Control Panel

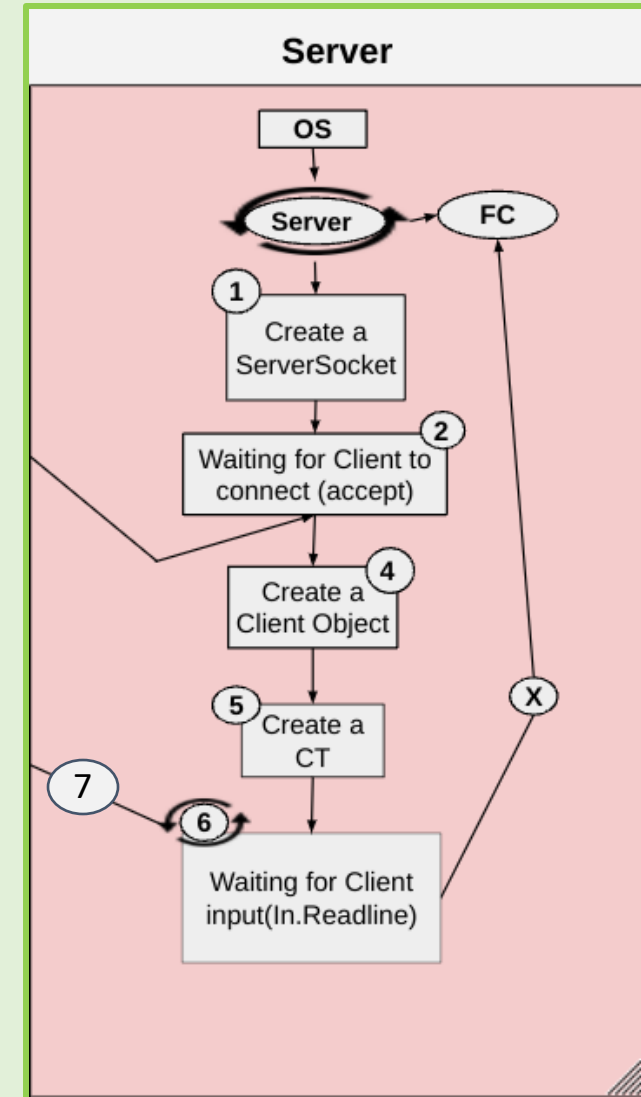
Client's Control Panel sends update request to Server's Client Worker
(updates change of values on sliders)



"C" messages (Update Field UI): Server side

Tool Tips:

- 2. Call accept of ServerSocket (block Server control)
- 5. Create a Client Thread
- 6. Client Thread waits for Client to send Control request
- 7. Client Thread sends values from FoodChain to Client
- X. Retrieves values requested by Client from FoodChain



"C" messages (Update Field UI): Client side

Tool Tips:

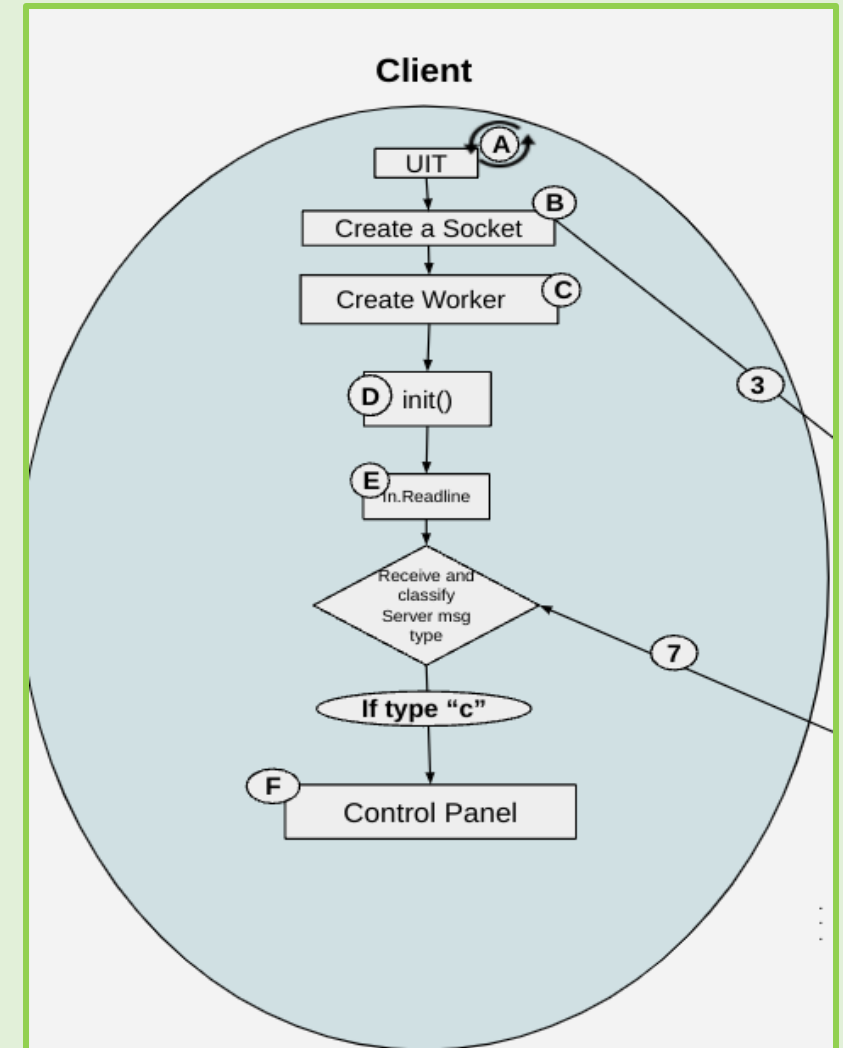
A: create a UI Thread

B: create a Socket (unblocks Server that was waiting for Client response)

C: create a Worker Thread for Client

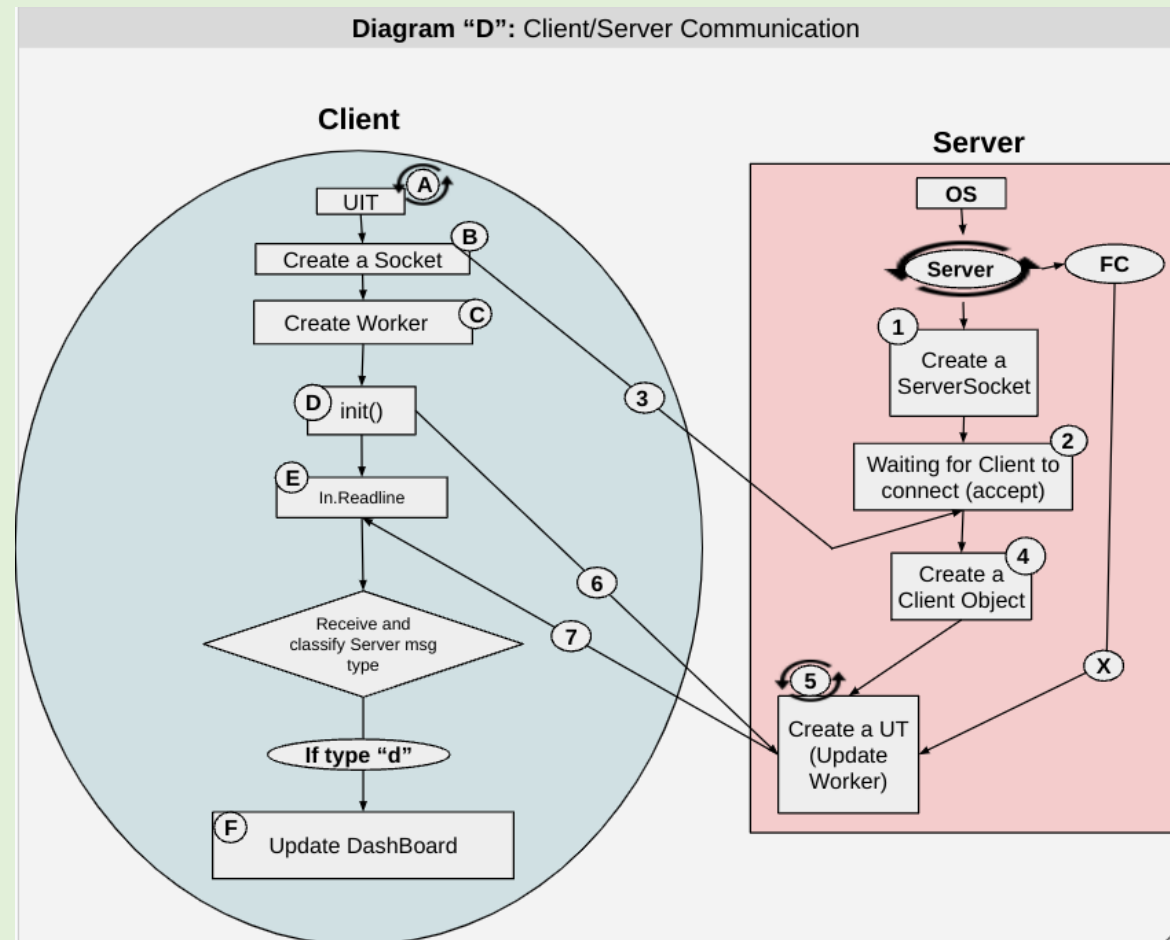
E: waiting for msg from Server (in)

F: Update Control Panel (values in sliders)



"D" messages: Update Control Panel

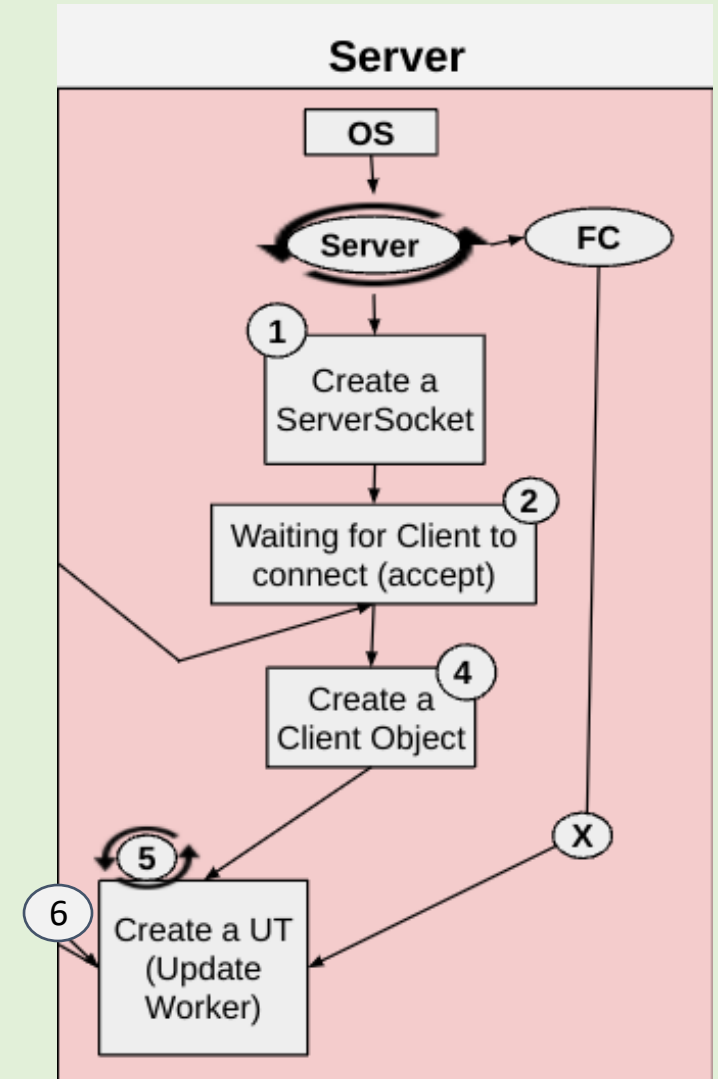
Client's DashBoard sends update request to Server's Client Worker (constantly pushing values displayed on DashBoard, population, weather, etc.)



"D" messages (Update Field UI): Server side

Tool Tips:

2. Call accept of ServerSocket (block Server control)
4. Create a Client Thread
5. Client Thread waits for Client to send Dashboard request
6. Client Thread sends Dashboard values from FoodChain to Client
- X. Retrieves values requested by Client from FoodChain



"D" messages (Update Field UI): Client side

Tool Tips:

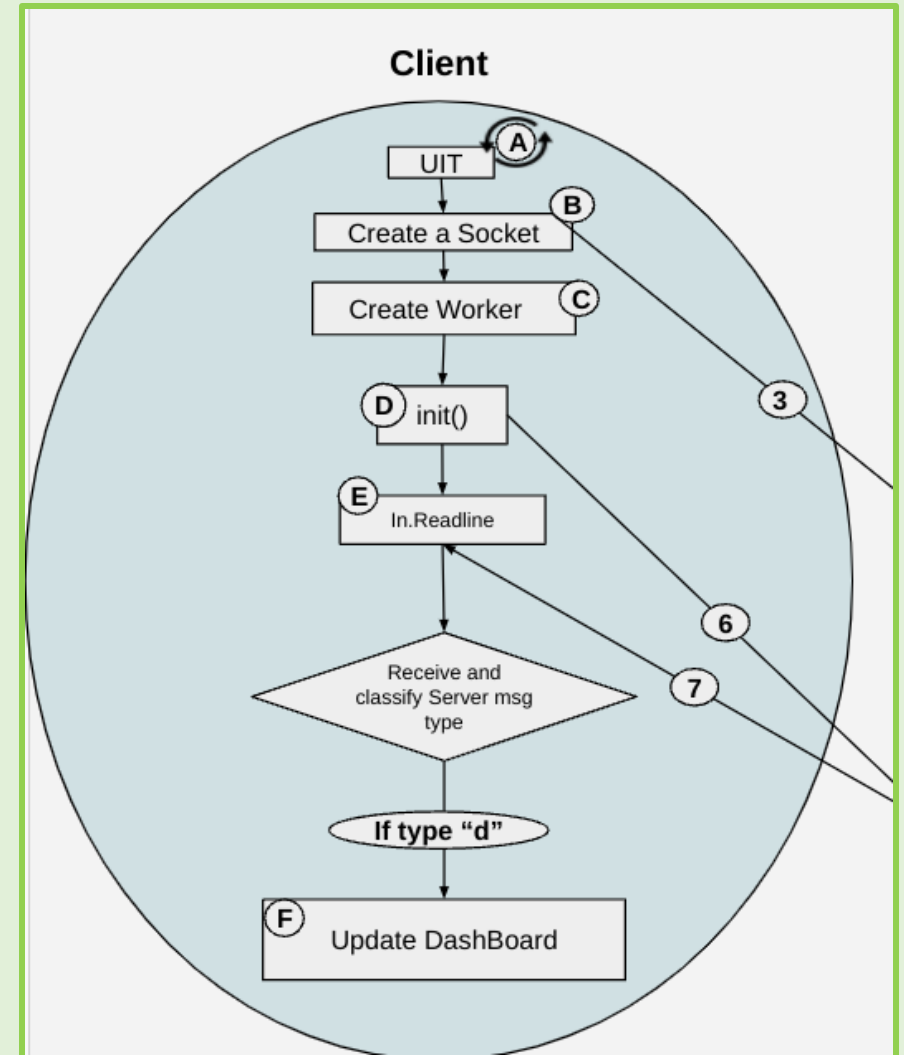
A: create a UI Thread

B: create a Socket (unblocks Server that was waiting for Client response)

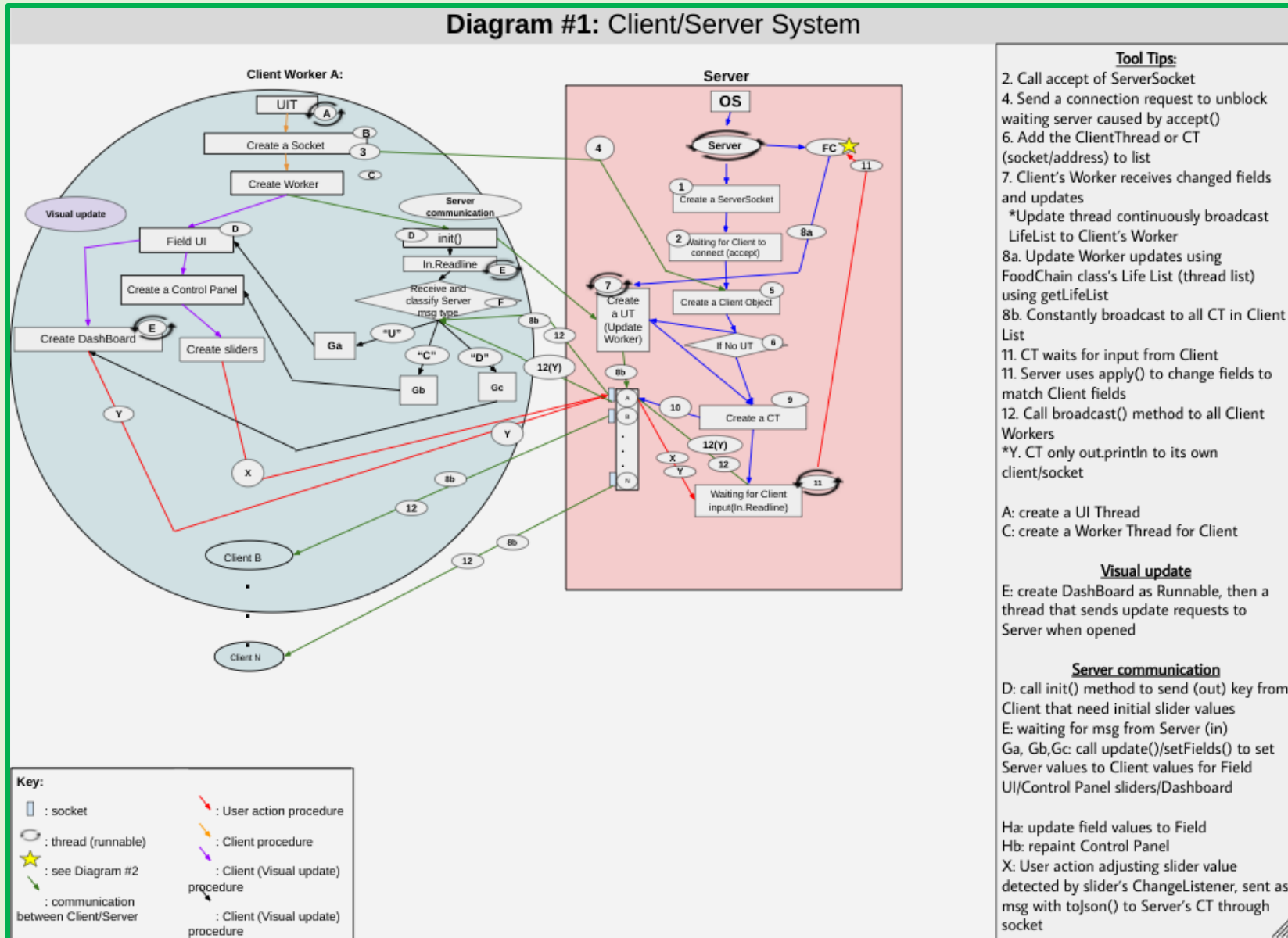
C: create a Worker Thread for Client

E: waiting for msg from Server (in)

F: Update Dashboard (values in labels)



The Whole Picture: Client/Server System



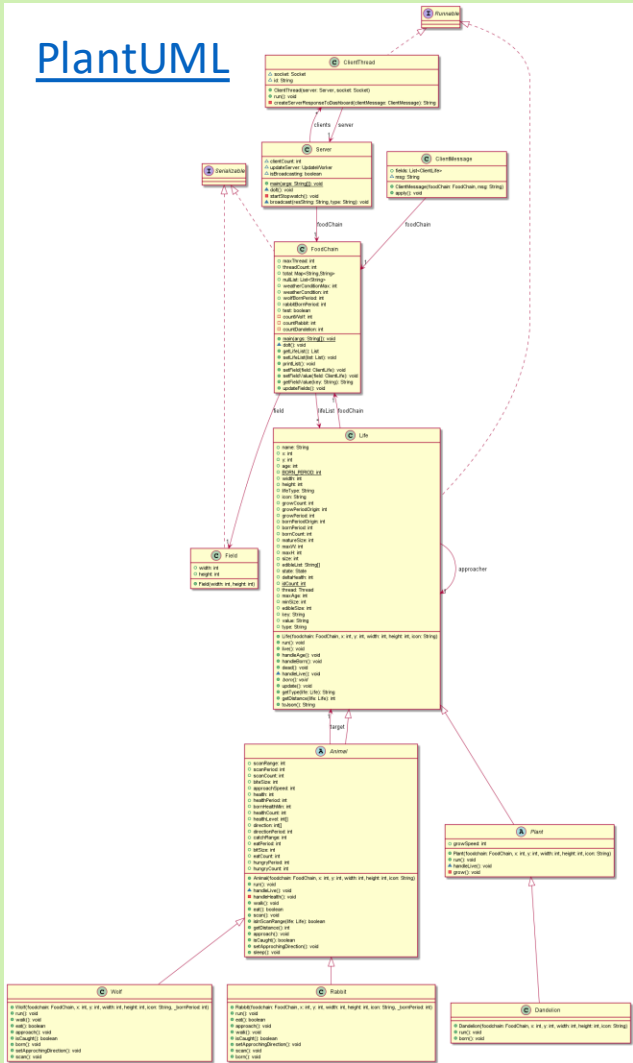
Java Application

OOP Programming

- Inheritance
- Encapsulation
- Polymorphism

Inheritance

The system of superclasses and subclasses in which subclasses can inherit superclasses' accessible fields and methods



WHY?

- less code needed; parent class code can be reused in child class
- reduce redundancy
- *cleaner code
- Overriding: methods with identical "signatures" in superclass and its subclass, but polymorphism narrows it down to lowest subclass

Inheritance examples

Dandelion class has
superclass of Plant
class

```
6
7 public class Dandelion extends Plant{
8
9     public Dandelion(FoodChain foodchain, int x, int y, int width, int height
10         super(foodchain, x, y, width, height, icon);
11
12         growSpeed = 2;
13         size = 10;
14         this.bornPeriod = 250;
15         Random rand = new Random();
16         this.bornCount = 20+rand.nextInt(Math.max(bornPeriod, 2));
```

With inheritance,
Dandelion class can
inherit and reuse Plant
class's constructor

Encapsulation

the process of wrapping or hiding data from other classes

WHY?

- maintenance of the code becomes easier and cleaner
 - *easy to change
- Setting fields and methods private is also important in efficiency.
 - *prevents other classes to access

Foodchain's List lifeList is encapsulated with the private access modifier

```
public class FoodChain implements Serializable{
    private List <Life> lifeList = new ArrayList <Life>();
    ...

    public void setField(ClientLife field) {
        if (field.value==null) {
            return;
        }

        if (field.key.equals("weatherCondition")) {
            weatherCondition = Integer.parseInt(field.value);
            //System.err.println("weatherCondition="+weatherCondition);
        } else if (field.key.equals("wolfBornPeriod")) {
            int newPeriod = Integer.parseInt(field.value);
            for (int i=0; i<lifeList.size();i++) {
                Life life = lifeList.get(i);
                if (life==null || !(life instanceof Wolf)) {
                    continue;
                }
                //int oldPeriod = life.bornPeriod;
                life.bornPeriod = newPeriod;
                //if (newPeriod<oldPeriod) {
                life.bornCount = 0;//Math.min(life.bornCount, life.bornPeriod);
                //} else {
                // life.bornCount = Math.max(life.bornCount, life.bornPeriod);
                //}
            }
            wolfBornPeriod = newPeriod;
        } else if (field.key.equals("rabbitBornPeriod")) {
```

Going through different (keys): able to use lifeList's values

```
public class ClientMessage {
    ...

    public void apply() {

        for (int i=0; i<fields.size(); i++) {
            ClientLife field = fields.get(i);
            foodChain.setField(field);
        }
    }
}
```

ClientThread class's apply() method does not know what happens in setField() or lifeList as they are encapsulated

Polymorphism

: an action with multiple forms

WHY?

- One variable name can be used to store variables of many data types

- Overloading: methods with different "signatures" (name, parameters, return type) in terms of parameters (different number, type, or both)

Ex. : `void A(int i)` vs `void A()` vs `void A(int i, int ii)`

- Overriding

```
public abstract class Life implements Runnable{
    public Life(FoodChain foodchain, int x, int y, int width, int height, String icon) {
```

Life.java

...

```
public void live() {
    while (state!=State.DEAD) {
        try{Thread.sleep(100);}catch(InterruptedException e){System.out.println(e);}
        handleAge();
        handleLive();
        handleBorn();
    }
}
```

Plant.java

```
public abstract class Plant extends Life {
    ...
    public Plant(FoodChain foodchain, int x, int y, int width, int height, String icon) {
        ...
    }
    void handleLive() {
        switch (state) {
            case GROW:
                grow();
                break;
            case DEAD:
                dead();
                break;
            case BORN:
                born();
                break;
            default:
                break;
        }
    }
}
```

```
public abstract class Animal extends Life {
    public Animal(FoodChain foodchain, int x, int y, int width, int height, String icon) {
```

Animal.java

...

```
void handleLive() {
    switch (state) {
        case NORMAL:
            walk();

            break;
        case SLEEP:
            sleep();
            break;
        case SCANNING:
            scan();
            break;
        case APPROACHING:
            approach();
            break;
        case EATING:
            eat();
            break;
        case DEAD:
            dead();
            break;
        case BORN:
            born();
            break;
        default:
            break;
    }
}
```

Abstraction

: reducing an object to its fundamentals in order to hide its attributes, methods, or communication

WHY?

- - To avoid implementation of methods for classes that we don't want an instance of
- Useful when wanting to force concrete classes to do individual implementation instead of using parent implementation
- Classes that are set abstract do not have to override/implement abstract methods from its parent class

Life (Animal, Plant as well) set to abstract to never be instantiated, so born() is set abstract

```
public abstract class Life implements Runnable{  
    ...  
    public abstract void born();  
}
```

Does not need to implement born() from parent class because of abstraction

```
public abstract class Animal extends Life {  
    ...  
}
```

Animal
extends
Life

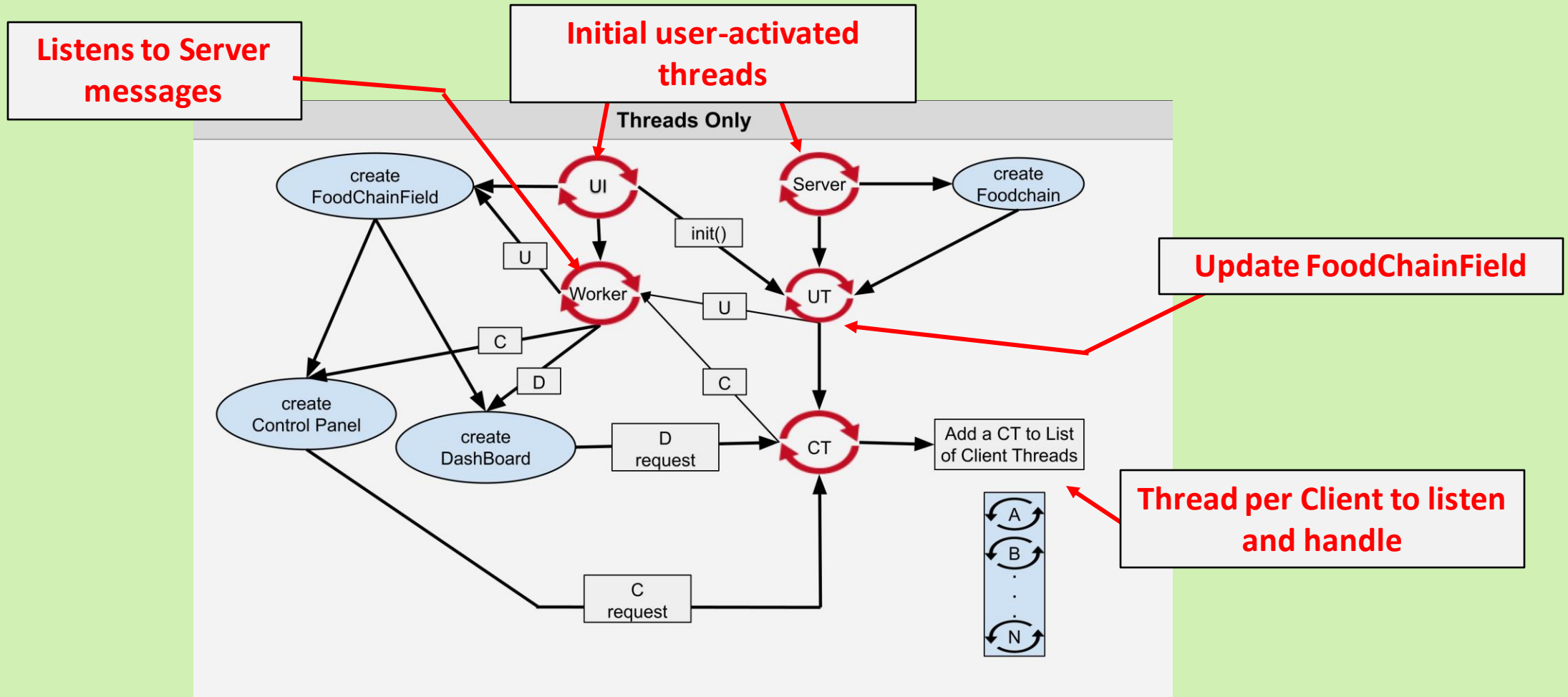
Since Wolf class is concrete, the class must implement abstract born() from its parent class

```
public class Wolf extends Animal{  
    ...  
    public void born(){  
        Life life = new Wolf(this.foodChain, this.x+1, this.y+1, maxW/2, maxH/2, icon, bornPeriod);  
        foodChain.addLife(life);  
        life.thread = new Thread(life);  
        life.thread.start();  
        foodChain.threadCount++;  
        state = State.NORMAL;  
        if (foodChain.test) {  
            System.err.println("born "+life.name+" from "+this.name);  
        }  
    }  
}
```

Wolf
extends
Animal

Multi-Thread

: concurrent actions (threads) executed by CPU; required for management in multi-socket Client systems (5 in this example)




Problem: Race Condition

: more than 1 thread race to perform the same action and end up overlapping/error

ex: CT and UT use same socket (PrintWriter out method) to communicate with client

Synchronized:
Only one thread can
use method at a
time, so
broadcasting
wouldn't overlap



Solution: Mutual exclusive, or not allowing two events happen at once

```
synchronized void broadcast(String resString, String type) {  
    try {  
        for (int i=0; i<clients.size(); i++) {  
            ClientThread client = clients.get(i);  
  
            PrintWriter out = new PrintWriter(client.socket.getOutputStream(), true);  
            out.println(resString);  
            //if (type=="c") {  
            //    System.err.println("Broadcasting to "+client.id+": "+resString);  
            //}  
        }  
    } catch (Exception e) {  
        System.out.println("Initializing error. Try changing port number!" + e);  
    }  
}
```


Appendix

- Ticks (Slide 4): an iteration through the class, Life's live() method