[CS209A-25Fall] Assignment 2 - QQ Farm

> **Question Design:** Yao Zhao
> **Code Sample:** Jinrun Liu

Deadline: 11:59 PM, Nov. 23

Evaluation: Demonstrations will be checked during the Week 12 lab session.

Late submissions will NOT be accepted after the deadline.

---

1. Overview

In this assignment, you will implement a simplified multiplayer version of **QQ Farm** using **Java Socket Programming**, **Multithreading**, and **JavaFX** for the graphical interface. Each player owns a small farm (4 × 4 grid) where they can **plant**, **harvest**, and **steal** crops from other players.

This project focuses on concurrent programming, thread safety, and synchronization among clients connected to a common server.

You can download demo codes from Sample Code Repository.

---

2. Key Requirements

2.1 Server-Side Implementation (35 points)

The server manages all players, crop growth, and concurrent stealing requests.
It must ensure that updates to shared data (e.g., crop yield) are **atomic** and **thread-safe**.

Responsibilities

- **Connection Management:**
  Accept multiple client connections; spawn one thread per client or use a thread pool.
- **Game State Management:**
  Maintain a `Farm` object per player (`4×4` plots).
  Each plot holds its crop state (`EMPTY`, `GROWING`, `RIPE`).
- **Crop Growth Threads:**
  Each planted crop should grow automatically (e.g., mature in 10 seconds).
  A background timer or thread periodically updates crop states.
- **Atomic Operations:**
  Implement synchronized logic for:
  - Planting (cannot overwrite growing/ripe crop)
  - Harvesting (owner collects full yield)
  - Stealing (reduce victim's yield → increase thief's coins)

2.2 Client & GUI Implementation (40 points)

Each client provides a JavaFX GUI that lets the user interact with their farm and others.

Responsibilities

- **Core GUI Layout:**
  Display a 4×4 grid of plots with clear visuals for empty/growing/ripe states.
  Surface player name, coin balance, and contextual status messages.
- **Action Controls:**
  Provide buttons for `Plant`, `Harvest`, `Visit Friends`, and `Steal`, with appropriate enable/disable feedback.
  Visiting another player should switch the displayed grid and retain navigation back to the home farm.
- **Visual Feedback:**
  Highlight plot state changes with icons, colors, or tooltips; show toast/status messages for success/failure cases (e.g., “Crop matured!”, “Steal failed”).
- **Networking & Updates:**
  Maintain an asynchronous socket connection, send player actions, and listen for server push notifications.
  Refresh the GUI in real time when crops mature, are harvested, or get stolen, using `Platform.runLater()` for all UI updates.
- **Responsiveness:**
  Ensure long-running tasks stay off the JavaFX Application Thread so the UI remains interactive during network activity.

2.3 Gameplay Rules

Action	Behavior
Plant	Costs coins and time; crop state becomes `GROWING`.
Harvest	Owner collects yield when `RIPE`.
Steal	Allowed only when target crop is `RIPE` and owner is away from their farm; each thief gets ≤ 25 % of yield; atomic update prevents over-stealing.
Visit Friends	Choose a friend and visit your friend's farm and switch the interface.

2.4 Concurrency & Synchronization (15 points)

- Implement server-side locking or compare-and-set logic when multiple clients target the same crop.
- Ensure background growth timers and network handlers do not block each other; document your threading model.
- Provide evidence (logs, tests, or demos) that simultaneous plant/harvest/steal requests keep the farm state consistent.

2.5 Exception Handling (10 points)

- Handle server crashes or disconnects gracefully with user notice.
- Validate user inputs (e.g., cannot plant on occupied plot).
- Catch network I/O errors without crashing the UI.

---

3. Evaluation Rubric

Category	Points	Focus Areas
Server	35	Thread safety, state consistency, and protocol design
Client	30	Networking logic, responsiveness, and usability
GUI	10	Visual clarity, feedback cues, and interaction design
Concurrency	15	Race-condition handling and synchronized operations
Error Handling	10	Graceful recovery from client/server/network failures

4. Suggested Implementation Steps

1. **Single-Player Version** – Implement plant → grow → harvest locally.
2. **Add Networking** – Connect client and server for state sync.
3. **Add Stealing** – Handle atomic updates and conflicts.
4. **Refine GUI + Error Handling**.

---

5. Demonstration & Self-Check

Complete the following checklist during your lab demo. The TA will walk through each checkpoint in order. Passing every checkpoint earns the demonstration credit; if any step fails, you can retry on the spot after fixing the issue.

5.1 Pre-Demo Preparation

- Start your server locally and ensure you can create or load at least three player accounts (needed for the multi-thief scenario).
- Successfully launched the client and provided the README for launching the client.
- Open a terminal or log viewer that shows server-side concurrency logs or debug prints; the TA must be able to see thread activity when you claim an operation is concurrent.

5.2 UI Walkthrough (Single Client)

- Launch one client and log in to your own farm. Point out the main UI regions: 4x4 plot grid, coin counter, action buttons, and status/notification area.
- Demonstrate planting a crop on an empty plot. Explain what visual indicator confirms the action (e.g., icon, color, tooltip).
- Wait for at least one crop to mature (or fast-forward using your testing tools) and show the automatic state transition from growing to ripe.
- Harvest the ripe crop. Confirm the coin balance increments and the plot resets to empty. Describe any confirmation messages the user sees.

5.3 Single-Client Robustness Checks

- Attempt to plant on an occupied plot and show that the client/server rejects the request with a clear message.
- Disconnect the network cable or disable Wi-Fi briefly (or simulate via `Disconnect` button if implemented) while the client is running. Demonstrate that the UI warns the player and remains responsive without crashing.
- Reconnect and show that pending operations (e.g., crop growth timers) continue or resynchronize correctly once the connection is restored.

5.4 Multi-Client Interaction

- Start a second client logged in as a different player. Show both farm UIs side-by-side.
- From Client A, visit Client B's farm. Confirm that plot states and coin counts match what Client B sees.
- Trigger a steal attempt from Client A when Client B's crop is ripe and B is "away" (per your game rules). Show the resulting state changes on both clients (yield reduction for victim, gain for thief) and any status messages.
- Demonstrate that the server broadcasts updates promptly: when Client B plants or harvests, Client A's view refreshes without manual reload.

5.5 Concurrency Stress Test (Multi-Thief Scenario)

- Launch a third participant (either a real client instance or a scripted bot). Prepare a ripe crop on Player C's farm.
- Have Clients A and B initiate a steal on the same ripe crop **at nearly the same time**. Use either:
  - a short script/automation that sends steal commands concurrently, or
  - coordinated manual input with a countdown, plus server logs showing overlapping timestamps.
- Show that the server handles the race condition correctly: the crop yield is deducted only once, each thief receives the appropriate share, and no negative or duplicate rewards appear. Highlight any log entries or on-screen notifications that confirm atomic handling.
- Repeat quickly if necessary to convince the TA that the behavior is consistent, not a one-off.

5.6 Threading & Responsiveness Verification

- While steals and crop updates are happening, interact with the GUI (mouse over plots, open menus) to prove the UI thread is responsive and not blocked by network operations.
- Display console logs or debugging information that identify worker thread names, demonstrating that long-running tasks are off the JavaFX Application Thread.
- If you implemented thread pools or timers, briefly explain their configuration and how you avoid deadlocks (e.g., order of locks, use of `Platform.runLater`).

5.7 Failure Recovery Scenarios

- **Client crash:** Force-close one client (kill the process). Show that the server detects the disconnect and releases any locks/resources. Relaunch the client, log back in, and demonstrate that the farm state remains consistent.
- **Server crash:** Terminate the server process while clients are connected. Each client should display a clear error message, disable invalid actions, and either retry connecting automatically or offer a "Reconnect" button. Restart the server and walk through the reconnection flow.
- Explain how unsent actions are handled when the crash occurs (e.g., queued requests are retried, discarded with a message, etc.).

5.8 Wrap-Up & Questions

- Summarize which parts of the rubric you have covered (server logic, client UI, concurrency, error handling).
- Point the TA to any additional documentation (README sections, diagrams) that clarifies your architecture or protocol.
- Be ready to answer questions about data consistency guarantees, thread synchronization choices, and how you would extend the system (optional but recommended).

6. Bonus

Outstanding demonstrations may be nominated for a Week 16 lecture showcase and will earn a +1 bonus toward the final course grade.

7. Submission Guidelines

Submit a ZIP file named `A2-yourStudentID.zip` containing:

- All source files and resources.
- `README.md` (architecture, run instructions, protocol description).

Upload the ZIP to Blackboard (BB) by **11:59 PM, Nov 23**.

Live demonstrations and grading will take place during Week 12 lab sessions. Be prepared to follow the checklist in Section 5.