Decentralized Tangram:
A game of co-operative tangram on distributed systems

Dafang Cao (j2d0b)
Edward Zhou (u6x9a)
Max Wei (e3v8)
Michael Chen (e6y9a)
Stephanie Chan (g2u9a)

1 Introduction

The tangram is a tiling puzzle game invented in China in the Song Dynasty (960-1279) and introduced to Europe in the early 19th century. It consists of seven flat shapes, called tans, and the objective of the puzzle is to put together the shapes without overlapping to form a new shape. In electronic form, it is a game where a player can select a shape, move it around and rotate it before placing it in its correct position.

2 System Description

The game exhibits interesting characteristics that we wanted to represent in the form of a distributed system. The tans themselves can be considered as resources which may only be held by one person at any given time. Essentially, they are mutually exclusive resources.

We represent the game as a fully connected network. Each node is a player, and every player may view and interact with the game state. Connecting to any client within the network will allow you to discover and connect to every other client. The allowed actions within the system which may be handled by any node include:

1. Joining the game
2. Picking up a tan
3. Modifying a tan
   a. via Rotation
   b. via Movement
4. Dropping a tan

Each action updates the game state. State is replicated across every node within the system. State is published to nodes immediately when they join. Each node must accept state updates they receive from other nodes. The state contains the following information:

1. The seven tangram pieces, each with:
   a. ID: A unique ID for the piece
   b. Shapetype: The type of tan it is (large triangle/ small triangle)
   c. Shape: SVG information, including:
      i. Points: The points for creating the path
      ii. Fill: The fill colour
      iii. Stroke: The stroke colour
   d. Player: The player currently controlling the piece
   e. Location: The coordinates of the piece on the board
   f. Rotation: The current rotation of the piece
   g. Clock: A logical clock which is updated on every action performed on the piece
2. A timer in ISO 8601 format with timezone UTC-07:00
3. A list of players, each with:
   a. ID: A unique ID for the player
   b. Name: A user-friendly identifier for the player
   c. Address: the address of the players RPC endpoint

3 Azure Usage

We will be running clients on the following servers. For each of these servers, port 8080 is exposed and used for the web server, while port 9000 is the RPC endpoint. All these servers will be connected.

Tang: https://52.160.86.26:8080
Qing: http://52.160.87.174:8080

Ming: http://52.160.80.243:8080
Yuan: http://52.160.85.192:8080

Note that they may be intermittently down prior to the demo.

4 Design Decisions

4.1 Joining

A node joins a network by specifying the address of an existing node and connecting to it, or by starting a new network (a new instance of a game). When connecting to an existing network, the joining node will retrieve a list of current active nodes from the specified node, and recursively apply this procedure as it connects to all remaining nodes. During this process, the respective lists of active nodes kept by each node is updated as new connections are made. In this way, nodes are able to maintain an agreement on the list of active nodes. This also prevents any issues from two nodes joining on two different existing nodes at the same time (the initial list of nodes they receive would not contain each other), as the recursive procedure will allow them to be made aware of each other. As soon as a connection is made, the joining node can receive state updates from the node it is connected to. A node must connect to all other nodes before any action can be performed, in the process bringing its state up-to-date and become subscribed to further updates.

4.2 Disconnections

Connectivity is checked between nodes using heartbeats. Every second, simple RPC calls are performed to check the status of peered nodes. Failing nodes will immediately be removed from the state, and to receive further state updates they must explicitly attempt reconnection with any node within the network.

4.3 Re-joining

Clients can rejoin immediately by re-establishing a connection with a node which is still alive. There is no distinction between a client rejoining versus a new client joining

4.4 Conflict Resolution

We use logical clocks to determine precedence of actions on tans. Each tan maintains its own lamport clock. A node can try to obtain lock on a tan when the tan has no current owner from its perspective. To obtain a tan, a node send a request to all peers, with an incremented logical timestamp. If every peer agrees that the logical timestamp in the request is greater than their current perceived time of a particular tan, the lock is obtained. Otherwise an action is rejected.

This algorithm is chosen over Ricart-Agrawala algorithm for a few reasons:

1. In our use case, a user would not interested in waiting for a tan to be released, instead an immediate rejection is more desirable. Ricart-Agrawala requires waiting for a resource to be freed.

2. One of the advantage of Ricart-Agrawala is the lack of release message. However release message serves a purpose in this projects UI, as we want to show the release of a tan in real-time.

4.5 Communication

We use RPC over TCP sockets to communicate between nodes and websocket to communicate between web front-end and web server. Communication between nodes use TCP connection despite the real-time constraints. This is because our mutual exclusion algorithm requires reliable connection. Websocket is adopted because it allows for real time communication without constantly opening new HTTP connections.

4.6 Trust

In our particular implementation, we chose to hold the assumption that all peers will not behave badly. Behaviour we considered bad include:

1. Connecting to only a subset of peers
2. Sending erroneous state updates
3. Locking tans indefinitely
4. Hijacking pre-existing IDs
5. Publishing different state to different peers
6. Locking more than one tan
7. Using different puzzle/solution sets

4.7 Time

For the timer, the initial node which sets up the game puts the game creation time into the state. Afterwards, clients simply calculate the game time by finding the difference between the current time and the start time.

A newly joined client establishes the start time with regards to its local clock by performing time synchronization with an existing client: The start time is calculated by subtracting local time by game length. Latency is compensated by approximating round trip time with timestamps. The granularity is within the second and appropriate for our use case: a visual indicator.

Over long periods of time the clocks might diverge because we do not perform further synchronization. The decision of not performing further synchronization is made due to the lack of an authoritative clock in a decentralized system. Another reason to avoid constant synchronization is that inaccuracy and overhead might lead to accelerated divergence.

It is assumed that a node and connected web frontends have externally synchronized clocks.

4.8 Public IP

We query an external service to determine our public IP (https://ipv4.wtfismyip.com/text). This is for convenience, so we do not have to manually pass in the IP for other clients to use.

5 System Setup

In this example, we will show the setup procedure of two nodes, X and Y.

Client X
1. X is launched via go run client.go -p 9000 :8080
2. config.json is read to determine information such as:
   a. The size of the board
   b. The tans and their shapes, sizes, location, and colours
   c. The solution
3. An RPC endpoint is opened under 9000
4. The node generates for itself a new ID
5. Given the configuration, state is instantiated
6. A heartbeats goroutine begins, which will check connections once any occur
7. A websocket endpoint is opened under :8081/ws
8. Static files are served via :8080/

Client Y
1. Y is launched via go run client.go -c :9000 -p 9001 :8081
2. An RPC endpoint is opened under 9001
3. The node generates for itself a new ID
4. A Node.Connect request is made to the specified address under the -c flag
5. The response contains config, state, and the player information of that node
6. Y synchronizes its state and time with Xs
   a. On state synchronization, Y also recursively connects with and synchronizes state with every other player
7. A heartbeats goroutine begins
8. A websocket endpoint is opened under :8081/ws
9. Static files are served via :8081/

6 RPC API Description

Node.Connect
Arguments
ConnectRequest
tangram.Player
ID tangram.PlayerID
Name string
Addr string
Returns
Config tangram.GameConfig
State tangram.GameState
Player tangram.Player
Custom Errors
An error occurs if the requesting client is already ingame

Connect requests that the remote node add the connecting node into the global state. The connecting node is then given the game config, the current state, and remote node player information.

Node.Ping
Arguments
tangram.PlayerID
Returns
bool
Custom Errors
None

Ping is used to check connectivity between nodes.

Node.GetTime
Arguments
int (unused)
Returns
int64
Custom Errors
None

GetTime returns the time since the game has started.

Node.LockTan
Arguments
LockTanRequest
Tan tangram.TanID
Player tangram.PlayerID
Time lamport.Time
Returns
bool
Custom Errors
Tan must be a valid tanID

LockTan checks for consensus from a given node whether it has the permission to take control of a tan. If the time supplied is after the current time on that tan, the action will succeed.

Node.MoveTan
Arguments
MoveTanRequest
Tan TanID
Location Point
Rotation Rotation
Time lamport.Time
Returns
bool
Custom Errors
Tan must be a valid tanID

MoveTan checks for consensus from a given node whether it has the permission to move or rotate a tan. If the time supplied is after the current time on that tan, the action will succeed.

7 Websocket API Description

We use gorilla websockets as our websocket library. Websockets are used in our application to publish events to and from the browser client.

7.1 Client-Sent

GetState
Requests state information from the node. This is primarily used for instantiation.

ObtainTan
Arguments
Tan tangram.TanID
Release bool

Signals to the server that the browser is attempting to lock or release a tan. The server is responsible for broadcasting this information. This occurs asynchronously and does not guarantee to the client that the action succeeded.

MoveTan
Arguments
Tan tangram.TanID
Location tangram.Location
Rotation tangram.Rotation

Signals to the server that the browser is attempting to move or rotate a tan. The server is responsible for broadcasting this information. This occurs asynchronously and does not guarantee to the client that the action succeeded.

7.2 Server-Sent

Player
Player is the current player's information and it includes the following:
1. ID: The randomly generated ID of a player when it first enters the game
2. Name: The name of the player
3. Address: The IP address and port of a player

Config
The config object contains the information to initialize a game.
1. Offset: This is the offset of the solution tans, which are created at 0,0 and translated by the x and y coordinates in the tans.
2. Size: This is the size of the boxed area that the tans are restricted to move in

3. Tans: Information about the tans and their initial whereabouts and rotation

4. Targets: The tans with the location and rotations of the solution

State

State contains the state information of the current game and it is synchronized throughout all the players. It has the following properties:

1. Host: The player who starts the game
2. Players: The players who are in the game
3. Solved: A boolean to indicate whether the puzzle is solved or not
4. Timer: A timer counting up from when the game is started
5. Tans: Information about the tans and their current whereabouts, rotation and time (using Lamport clocks)

8 Unimplemented / WIP

Host Selection

In the proposal, we proposed to use an elected host as an optimization to reduce the amount of required messaging, also creating a star topology. The current implementation has no such mechanism and we have only a fully connected topology.

UI Improvements

There are various UI improvements such as cleaner UI, smoothing, and general tweaks for UI that can be implemented. In particular, we are missing the following features:

1. Indication that locking on a tan has failed.
2. Notification to inform the player if they have disconnected.
3. Solved puzzle indicator.
4. Player list that updates real-time as players connect/disconnect.

Tan Lock Timeout

If a client who has obtained a tan disconnects, there is no lock release.

NAT Traversal

The current implementation does not allow nodes in different private networks to connect. Adjustments to accommodate for nodes in private networks using NAT discovery is being developed. While a working prototype using go-libp2p-nat library is available in one of the branches, it fails in certain testing environments.