Watch Out

DPS Lab Project - July 2024

Admin Server: Players

- The admin server maintains a list of all the registered players.
  - We use a ‘Players’ thread safe singleton.
- Multiple threads may access this list concurrently.
  - This is due to possibly simultaneous requests coming to the server.
- We may want to concurrently:
  - Check whether or not a player is already registered.
  - Register a new player.
  - Get the list of all registered players.
- We protect access to the data structure using synchronized methods of the ‘Players’ singleton.
  - When returning the list of all registered players, we actually return a **copy** of the list.

Admin Server: Heartbeats

- The admin server maintains a list of all the heartbeat data points.
  - We use a ‘Heartbeats’ thread safe singleton.
- Multiple threads may access this list concurrently.
  - This is due to possibly simultaneous requests coming to the server.
- We may want to concurrently:
  - Add some new data points to the list.
  - Compute some statistics over the list.
- We protect access to the list using the list’s intrinsic lock.
- When adding new data points to the list, we first acquire the list’s lock and then release it only after all the additions are done.

Admin Server: Heartbeats Statistics

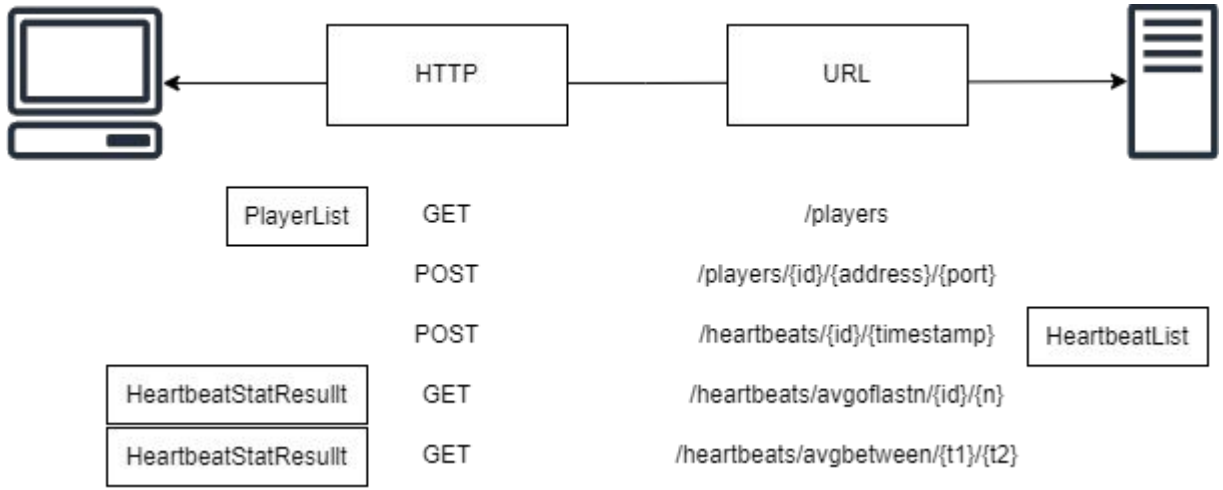
When computing statistics on the list:

- We acquire the list's lock.
- Iterate over the list, copying all the data points necessary for the computation into another thread local temporary list.
- We release the list's lock.
- We do the possibly lengthy computation on the thread local temporary list.

Admin Server: Synchronization Considerations

- We have **two** different **independent** singletons:
  - One for the players data structure.
  - Another for the heartbeats data structure.
- We tried to achieve a synchronization as fine grained as possible.
  - For example, we can add a new player while heartbeat statistics are being computed.
- We can also add **new heartbeat data points while computing statistics**.
  - We **only block when copying** the relevant data points to the thread local list.

Admin Server: REST API



Insertion of a Payer in the Network

As soon as a new Player process gets started:

- The process registers itself to the Admin Server as a new player.
- The player receives back the list of all the already registered players.
  - For each player, we have its id, its address, its port and its starting position on the pitch.
- The player broadcasts a greeting message to all the already registered players.
  - The greeting message contains the player's id, the player's address, the player's port and the player's starting position on the pitch.
- By doing this, we know that **each player will eventually come to know all other players** that are participating in the game.

Design of the P2P Network

- Each player maintains a list of all the other participating players.
- Thus, each player is able to directly communicate with all other players.
  - We use asynchronous unary gRPCs for all communications among players.
- On top of this list we define a **ring overlay network**.
- This overlay network is used for implementing the **election** and **mutual exclusion** algorithms.
- The ring is defined as follows:
  - The successor of a player p, with id k, is the player p' whose id is the minimum of the set of all player ids that are **greater than k**.
  - If such minimum does not exist (i.e. k is the highest id), then the successor of p is the player p' whose id is the minimum of the set of **all player ids**.

Seeker Election Algorithm: Game Start

- The algorithm is based on Chang and Roberts ring election.
- The algorithm starts as soon as a player receives a **game start notification** through MQTT.
- The notification is processed only if the player is Idle. Otherwise it is ignored.
- If the player considers itself a possible seeker:
  - It moves itself to a Voted state.
  - It sends an **election message** to the next player along the ring.
- The election message contains:
  - The player's id.
  - The player's starting position on the pitch.
- A player considers itself a possible seeker only if it is the closest to the home base, compared to all other players **known** to it.

Seeker Election Algorithm: On Election Receive

When a player receives an **election message**:

- If it is Idle, the player:
  - Moves itself to a Voted state.
  - It either forwards the election message or sends a new election message depending on who is a better candidate between itself and the one in the received election message.
- If it has already Voted:
  - If the received election message is the one that was previously sent by the player:
    - It means that the player is the Seeker.
    - The player sends a seeker message along the ring.
    - The **seeker message** contains the seeker's id.
  - If the received election message is not the one that was previously sent by the player:
    - If the candidate in the message is better than the player, the player forwards the election message along the ring.
    - If the candidate in the message is worse than the player, the player does nothing, blocking the election message.
- In **any other state**, the player does nothing, blocking the election message.

Seeker Election Algorithm: On Seeker Receive

When a player receives a **seeker message**:

- If it is Idle:
  - The player missed its chance to vote and it is a Hider.
  - The player forwards the seeker message along the ring.
- If it has already Voted:
  - The seeker message cannot be its own. The player is thus a Hider.
  - The player forwards the seeker message along the ring.
- If it is the Seeker:
  - It must be that the id bundled together with the seeker message is the same as the player's id.
  - It means that the seeker message went around the ring.
  - The player starts **pursuing** the other players.
  - The player starts the **token ring mutual exclusion algorithm** by sending a **token** along the ring.

Seeker Pursuit

- The pursuit logic is executed by a **new thread**.
- This thread is spawned as soon as the seeker message does a trip around the ring, coming back to the sender.
- The pursuit logic is as follows:
  - The Seeker keeps track of the set of all **taggable players** for this round of Watch Out.
    - It clears this set.
    - Then, it initializes it with the ids of all other players known to it.
  - While this set is not empty
    - It finds the closest taggable player to it.
    - It **waits** for an amount of time equal to the time required for it to reach said player.
    - After this amount of time has passed, if the pursued player is still taggable:
      - It sends a **tag message** to it.
      - It removes the player from the set of **taggable players**.
  - Once the set of taggable players is empty, the thread's execution is over.

Token Ring Mutual Exclusion Algorithm (1)

When a player receives the **token**:

- If it is Idle.
  - It has skipped the election. It is a Hider.
  - Do the same thing a Hider would do (read below).
- If it is a Hider:
  - It tries to **go for the home base**.
  - Once that is done, it forwards the token along the ring.

Hider Going for the Home Base

- The Hider **waits** for an amount of time equal to the time required for it to reach the home base.
- After the **wait** expires, we either may have been tagged or not.
- Thus, the Hider **checks its own state**.
- If it is still a Hider, it means that it hasn't been tagged yet.
  - It thus becomes Safe.
  - **Waits** for ten seconds.
  - After this amount of time has passed, it signals to all other players the fact that it is Safe by broadcasting a **round leave message**.
- If it has been Tagged during the wait, it does nothing.

Token Ring Mutual Exclusion Algorithm (2)

When a player receives the **token**:

- If it is a Seeker:
  - If there are still **taggable players**, it forwards the token along the ring.
  - If there are no more **taggable players**:
    - It blocks the token.
    - It becomes Idle
    - It signals to all other players that the current round is over by broadcasting a **round end message**.
- If it either is Safe or Tagged:
  - It forwards the token to the next player.

On Tag Receive

When a player receives a **tag message**:

- If it is Idle:
  - It becomes Tagged.
  - It broadcasts to all other players a **round leave message**.
- If it is a Hider:
  - It becomes Tagged.
  - It broadcasts to all other players a **round leave message**.
  - It **notifies** any possible waiting thread (i.e. the thread going for the home base).
- If it is Safe, it does nothing.

On Leave Round Receive & On Round End Receive

When a player receives a **round leave message**:

- It removes from the set of **taggable players** the sender of the message.

When a player receives a **round end message**:

- If the player is either Idle, Hider, Safe or Tagged:
  - It becomes Idle.
- Any other state is not contemplated.

Player Peer Synchronization Issues (1)

- All the payer's shared state is bundled into a '**Context**' singleton.
- All the messages coming to a player are handled by auxiliary threads.
  - When receiving a message **Msg**, the auxiliary thread handles it by invoking a **onMsgReceive** method on the **Context** singleton.
- We may have multiple threads concurrently accessing the player's shared state.
  - We need to protect access to this shared state.
  - We do this by making as **synchronized** all the public methods of the **Context** singleton.
- This means that each auxiliary thread that handles a message:
  - First has to acquire the intrinsic lock of the **Context** singleton.
  - Only once the lock has been acquired, the message handling logic gets executed.
  - As soon as the handling logic has finished execution, the lock is released.
- This effectively translates to the fact that the player handles **one message at a time**.
  - There are some exceptions.

Player Peer Synchronization Issues (2)

- The first exception is when the player receives a custom text message from the admin client through MQTT.
  - In this case, we don’t need to access any shared state.
- The second exception is when the player **goes for the home base**.
  - This happens when the player receives the **token** and is either Idle or a Hider.
  - In this scenario, the message handling thread calls **wait**.
  - The **Context** singleton intrinsic lock is released during the **wait**.
  - Thus, other messages can be handled while the thread is waiting.
  - There are actually two calls to **wait**.
    - The first one can be **notified** by another thread handling a received **tag message**.
      - In this way, if the player gets tagged, it wakes up immediately, instead of waiting until the time out.
    - The second one is never **notified**.

Player Peer Synchronization Issues (3)

- The Seeker player pursuit logic, for a given round, is executed by spawning a new thread that runs the **seekOtherPlayers** method belonging to the **Context** singleton.
- Since this method needs to access the player's shared state, which can be modified by other concurrent threads, we need to mark is as **synchronized**.
- As we have stated earlier, **seekOtherPlayers** periodically calls **wait**.
- The **Context** singleton intrinsic lock is released during the **wait**.
- In this way, the Seeker is able to concurrently execute:
  - The pursuit logic.
  - Any other message handling logic.
- This **wait** is never **notified**.

Player Heart Rate Simulator and Sender

- As soon as the player successfully registers itself with the admin server:
  - It starts a **heart rate simulator** thread.
  - It starts a **heart rate sender** thread.
- The heart rate simulator and sender threads share a **buffer**.
- The **simulator** thread acts as a **producer**: it puts data into the **buffer**.
- The **sender** thread acts as a **consumer**: it removes data from the **buffer**.
- We must protect access to this shared buffer.
- The buffer is also the component that implements the **sliding window** technique.

Player Heart Rate Buffer

- The buffer manages **two** lists:
  - The list of raw data points, produced by the heart rate simulator thread.
  - The list of aggregated data points, consumed by the heart rate sender thread.
- The buffer offers two methods:
  - **addMeasurement**.
  - **readAllAndClean**.

Player Heart Rate Buffer: addMeasurement

When adding a new raw data point:

- We first acquire the intrinsic lock of the raw list.
- We add the new data point to it.
- If it is possible to compute a new aggregated data point:
  - We compute the aggregated data point.
  - We acquire the intrinsic lock of the aggregated list.
  - We add the newly computed aggregated data point to said list.
  - We release the intrinsic lock of the aggregated list.
  - We remove the first four data points in the raw list.
- We release the intrinsic lock of the raw ist.

Player Heart Rate Buffer: readAllAndClean

When reading the aggregated data points in the buffer:

- We acquire the intrinsic lock of the aggregated list.
- We make a copy of said list.
- We clear the aggregated list, removing all data points in it.
- We release the lock.
- We return the copy we made of the aggregated list.

Heart Rate Buffer: Synchronization Considerations

- When the heart rate sender thread reads data points from the buffer:
  - Only the aggregated list's intrinsic lock needs to be acquired.
- When the heart rate simulator thread adds data points to the buffer:
  - Most of the time, we only acquire the raw list's intrinsic lock.
  - When we need to acquire also the aggregated list's intrinsic lock, we do it just to add the newly computed aggregated data point, and then immediately release it.
- The producer and consumer threads block only when:
  - They write to and read from the buffer at the same time
  - The write leads to computing a new aggregated data point.