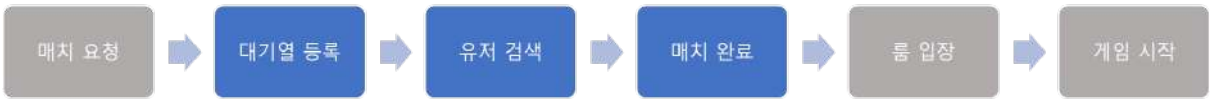
C# - Redis 를 이용한 매치메이킹

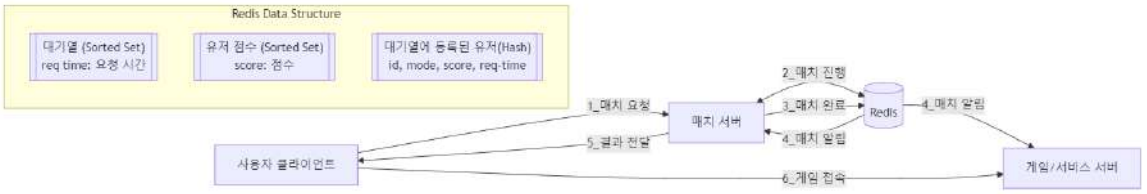
개요



매치 프로세스 중 [대기열 등록] - [유저 검색] - [매치 완료] 기능을 구현하였습니다

- 매칭 시스템은 사용자 간의 매치를 Redis를 사용하여 대기열 등록 및 검색을 관리하는 시스템입니다.
- 1인 매치부터 최대 5인까지 매치 가능하며, 매치 시간을 관리하고, 매치 평균 시간을 계산하여 빠른 매치가 가능하도록 하는 기능이 포함되어 있습니다.
- 주요 기능으로는 매치 대기시간 확인 및 MMR 조정, 평균 매치 시간 확인 후 Task 증가/감소, pub/sub 을 이용한 매치 요청, 완료 이벤트 연결 등이 포함됩니다.

매치 진행 플로우



1. **매칭 요청**

   사용자 또는 클라이언트가 매칭을 요청합니다. 매칭 요청은 레디스의 대기열에 등록되며, 요청은 해당 게임 모드와 조건에 맞춰 큐에 저장됩니다.

2. **매치 진행**

   매치 서버는 대기 중인 유저들 중 조건에 맞는 유저들을 검색하고 매칭을 진행합니다. 이 과정에서 MMR, 대기 시간 등 다양한 요소를 고려하여 가장 적합한 유저를 찾습니다.

3. **매치 완료**

   레디스에 저장된 매칭 대기열 데이터를 기반으로 유저들을 검색합니다. 매칭에 성공한 유저들은 레디스에서 큐에서 제거되고, 매칭이 완료됩니다.

4. **레디스의 pub/sub을 이용한 매칭 완료 알림**

   매칭이 완료되면 레디스의 pub/sub 기능을 통해 매칭 완료 이벤트를 발행합니다. 이를 통해 모든 관련 시스템과 클라이언트가 매칭 완료 상태를 실시간으로 알 수 있습니다.

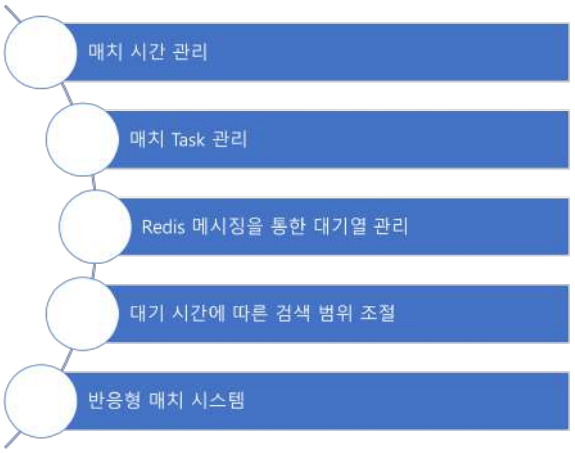
5. **클라이언트에게 매칭 완료 반환**

매칭이 완료되면 클라이언트에게 해당 매칭 결과를 전달합니다. 클라이언트는 매칭된 상대와 함께 게임을 시작할 준비를 합니다.

6. **클라이언트의 게임 서버 연결**

   클라이언트는 매칭 완료 정보를 받은 후, 게임 서버에 연결하여 게임을 시작합니다. 게임 서버는 매칭된 플레이어들이 함께 게임을 진행할 수 있도록 설정됩니다.

주요 기능



1. **매치 시간 관리**
   a. 각 매치 완료 시 매치 시간을 기록하고, 이 데이터로 평균 매치 시간을 계산합니다. 평균 매치 시간이 길어지면 매칭 대기열에서 검색 범위를 확대하여 더 많은 후보를 탐색함으로써 매칭 성공률을 높입니다.
2. **매치 Task 관리**
   a. 현재 매치 대기열 상태와 평균 매치 시간을 기준으로 작업 수를 동적으로 관리합니다. 매칭 대기열이나 평균 매치 시간이 길어지면 작업을 추가하고, 짧아지면 작업을 줄입니다. Task는 지정된 최소 및 최대 한도 내에서 조정되며, 변경 후 일정 쿨다운 시간을 적용해 빈번한 변동을 방지합니다.
3. **레디스 메시징을 통한 실시간 매칭 대기열 관리**
   a. Redis Pub/Sub 채널을 통해 매칭 대기열 상태를 지속적으로 구독합니다. 새로운 매칭 요청이 추가되거나 제거될 때마다 이를 실시간으로 반영합니다.
4. **대기 시간에 따른 대상 검색 범위 조절**
   a. 대기 시간이 길어질수록 매칭 후보자가 부족할 가능성이 높아집니다. 따라서 검색 범위를 넓혀 더 많은 후보자를 탐색합니다. 이를 통해 매칭 성공률을 높이고, 오래 기다린 유저들에게도 매칭 기회를 제공합니다.
5. **반응형 매칭 시스템**
   a. 새로운 매치 대기 요청이 등록되거나 기존 대기 요청이 해제될 때마다 시스템이 즉각 반응하여 상태를 업데이트합니다.

주요 기능 상세

평균 시간 관리

매칭의 평균 대기 시간을 효율적으로 관리하기 위해 고정된 수의 최신 매치 시간을 유지합니다.

`AddMatchTime` 메서드는 새로운 매치 시간을 추가하면서 오래된 데이터를 덮어쓰며, 최신 매치 시간들의 합을 유지해 계산 속도를 높입니다. `GetAverageMatchTime` 메서드는 현재 시간을 기반으로 유효한 매치 시간 데이터의 평균을 반환하고, 일정 시간이 지나면 데이터를 초기화하여 오래된 기록이 누적되지 않도록 합니다. 이러한 방식은 메모리 사용을 최소화하면서 실시간 평균 매칭 시간을 빠르게 제공해, 시스템의 성능을 최적화합니다.

```
public int GetAverageMatchTime()
{
  lock (_lock)
  {
    if (_totalSeconds == 0)
        return 0;

    if ((TimeHelper.GetUnixTimestam
p() - _lastAddedMatchTime) > WaitTi
meReset)
        ResetMatchTimes();

    int actualCount = _matchTimes.C
ount(t => t > 0);
    return actualCount == 0 ? 0 : _
totalSeconds / actualCount;
  }
}

private void ResetMatchTimes()
{
  _totalSeconds = 0;
  _currentIndex = 0;
  _lastAddedMatchTime = 0;
  Array.Clear(_matchTimes, 0, _matc
hTimes.Length);
}

public void AddMatchTime(int second
s)
{
  lock (_lock)
```

Task 관리

Task 관리 기능은 동시 작업 수를 효율적으로 조정해주는 역할을 합니다.

주요 기능인 `IncreaseTask`와 `DecreaseTask`를 통해 필요한 작업을 생성하거나 오래된 작업을 삭제하여 최적의 리소스 사용을 보장합니다. 또한, 쿨다운 함수인 `IsInCooldown`과 `UpdateCooldown`을 통해 작업의 생성과 삭제가 일정 시간 간격을 두고 실행되도록 제한해, 자원의 불필요한 낭비를 방지하고 시스템 안정성을 유지합니다.

```
public async Task IncreaseTask(T
Enum taskType, Func<object, Canc
ellationToken, Task> taskAction,
object param)
{
  if (GetTaskCount(taskType) >=
Constant.MaxTaskCount)
      return;

  if (IsInCooldown(taskType))
      return;

  var cts = new CancellationToke
nSource();
  var task = Task.Run(() => task
Action(param, cts.Token), cts.To
ken);

  var taskWithCancellation = new
TaskWithCancellation(task, cts);

  _tasks.AddOrUpdate(
     taskType,
     _ => new ConcurrentQueue<Ta
skWithCancellation>(new[] {
      taskWithCancellation }),
     (_, queue) => {
         queue.Enqueue(taskWithC
ancellation);
         return queue;
     });

  UpdateCooldown(taskType);
```

```
{
    _totalSeconds -= _matchTimes[_c
urrentIndex];
    _matchTimes[_currentIndex] = se
conds;
    _totalSeconds += seconds;
    _currentIndex = (_currentIndex
+ 1) % WaitTimeMaxCount;
    _lastAddedMatchTime = TimeHelpe
r.GetUnixTimestamp();
  }
}
```

```
  await Task.CompletedTask;
}

public async Task DecreaseTask(T
Enum taskType)
{
  if (GetTaskCount(taskType) <=
Constant.MinTaskCount)
      return;

  if (IsInCooldown(taskType))
      return;

  if (!_tasks.TryGetValue(taskTy
pe, out var queue) || queue.IsEm
pty)
      return;

  if (!queue.TryDequeue(out var
taskWithCancellation))
      return;

  taskWithCancellation.Cancellat
ionTokenSource.Cancel();
  taskWithCancellation.Cancellat
ionTokenSource.Dispose();
  await taskWithCancellation.Tas
k;

  UpdateCooldown(taskType);
}
```

대기 시간에 따른 검색 범위 조절

사용자의 대기 시간과 시스템 평균 대기 시간에 따라 MMR 검색 범위를 조절해주는 역할을 합니다.

평균 대기 시간이 길어질수록, 그리고 개별 사용자의 대기 시간이 길어질수록 가중치를 높여 더 넓은 범위의 다른 사용자를 검색할 수 있게 합니다. 평균 대기 시간과 사용자 대기 시간 각각에 대해 범위별로 다른 가중치를 적용하여, 매칭이 신속히 이루어질 수 있도록 합니다.

레디스 메시징을 통한 실시간 매칭 대기열 관리

Redis 메시징을 사용해 매칭 대기열 변화를 실시간으로 관리합니다. 구독 메커니즘을 통해 매칭 요청과 완료 이벤트를 수신하면, 해당 이벤트에 따라 대기열 수를 증가 또는 감소시킵니다.

이를 통해 매칭 대기열에 새 요청이 들어오거나 매칭이 완료될 때마다 즉시 반응할 수 있어 시스템의 실시간성을 높입니다. 메시지 발행 기능을 활용해 다른 인스턴스에서도 즉시 대기열 상태가 반영되도록 하여, 분산 환경에서 일관된 매칭 상태를 유지하도록 합니다.

```
public static long EncodeScore(int mmr)
{
  var ts = TimeHelper.GetUnixTimestamp();
  // timestamp * 10000 + mmr
  // 예: 1500 MMR, 1000초 → 10000000 + 1500 = 10001500
  return ts * MMR_MULTIPLIER + mmr;
}

public static (int mmr, int waitTime) DecodeScore(long score)
{
  // 하위 4자리는 MMR
  var mmr = (int)(score % MMR_MULTIPLIER);
  if (mmr < MIN_MMR || mmr > MAX_MMR)
      return (0, 0);

  // 상위는 등록 시간(초)
  var beginTime = score / MMR_MULTIPLIER;
  var waitTime = (int)(TimeHelper.GetUnixTimestamp() - beginTime);

  return (mmr, waitTime);
}

public int GetAdjustMMR(int waitTime)
{
  double averageMatchTime = GetAverageMatchTime();

  // 전체 대기 시간에 따라 가중치에서 30초까지 10%, 60초까지 20% 그외 25% 적용
  double processWaitWeight = averageMatchTime switch
  {
      >= 0 and <= 30 => (double)(WaitTimeWeight * 0.1), // 100
      > 30 and <= 60 => (double)(WaitTimeWeight * 0.2), // 200
      _ => (double)(WaitTimeWeight
```

```
public event OnMatchRequest? IncreaseMatchQueueEvent;
public event OnMatchComplete? DecreaseMatchQueueEvent;

private void InitSubMatchQueue()
{
_pubsub.SubscribeAsync(GetChannel(RedisKeys.MatchRequest), (channel, value) => {
  if (IncreaseMatchQueueEvent is null && !value.HasValue)
      return;
  IncreaseMatchQueueEvent?.Invoke(Converter.ToMatchMode(value.ToString()));
    });

_pubsub.SubscribeAsync(GetChannel(RedisKeys.MatchComplete), (channel, value) => {
  if (DecreaseMatchQueueEvent is null && !value.HasValue)
      return;

  var data = value.ToString().Split(':');
  var mode = Converter.ToMatchMode(data[0]);
  var count = int.Parse(data[1]);

  DecreaseMatchQueueEvent?.Invoke(mode, count);
    });
}

public void PubIncreaseMatchQueue(MatchMode mode)
{
  _pubsub.Publish(GetChannel(RedisKeys.MatchRequest),

Converter.ToFastString(mode));
}

public void PubDecreaseMatchQueue(MatchMode mode, int count)
```

```
* 0.25),              // 250
  };

  // 사용자 대기 시간에 따라 가중치
  // 15초까지 10%, 30초까지 20%, 40초까지 30%, 그외 40% 적용
  double waitTimeWeight = waitTime switch
  {
      >= 0 and <= 15 => (double)(WaitTimeWeight * 0.1), // 100
      > 15 and <= 30 => (double)(WaitTimeWeight * 0.2), // 200
      > 30 and <= 40 => (double)(WaitTimeWeight * 0.3), // 300
      _ => (double)(WaitTimeWeight * 0.4),              // 400
  };

  return (int)(processWaitWeight + waitTimeWeight);
}
```

```
{
  _pubsub.Publish(GetChannel(RedisKeys.MatchComplete),

$"{Converter.ToFastString(mode)}:{count}");
}
```