Parallel Programming Assignment 1

Ding Zhu, 995668157
Yi Li, 1001251860
Xinyi Lin, 1001091132

1. Introduction

This report is based on an assignment which is aimed at implementing and evaluating one of the dynamic load balancing algorithms: *Lightest* or *Spread*, present in [1] upon using SimMud, a massive multiplayer game simulator. Our group chose to implement the *Spread* algorithm. The rest of this report is organized as follows: Section 2 introduces the related background of SimMud and *Spread* algorithm. In section 3, we describes the design and the implementation of the load balance algorithm. Section 4 analyzes the performance of the static vs. dynamic schemes. Section 5 concludes this report.

2. Related Background

SimMud is a massive multiplayer game (MMO) simulator designed to emulate players, requests, maps, game objects and network traffics that happen in a real online game. Its simulation power simplifies the design and evaluate algorithms for online games and allows us to study unique characteristics of MMO games.

In terms of architecture, SimMud uses a classic client-server design where client handles graphic rendering and is equipped with AI to simulate player behaviors. Server, on the other hand, is a multithreaded application for handling authentication and responding to player requests. The original design of SimMud server incorporates a static load balancing algorithm which simply divide the game world into a grid of adjacent areas called Regions. Each regions is stored in a two-dimensional array and is assigned to a thread in a round-robin fashion. Players within a region will be handled by a single thread.

The challenge of static balancing is player flocking where many players are gathered into a single region. In another word, when players move from different regions to a single, the thread handling the single region will be heavily loaded while other threads are idle.

To meet the challenge, we designed a *Spread* algorithm to uniformly distribute workloads to available resources (e.g. threads). In ideal case, the *Spread* algorithm minimizes overall thread idle time and thus maximize performance.

3. Spread Algorithm Design and Implementation

Spread algorithm aims at distributing workloads equally to all threads. In SimMud, workloads are actually coming from user requests (e.g. join/leave/move/action) since all other objects such like walls or food won't move themselves. Therefore, to achieve equal workloads, we just need to equally distribute players to threads. Figure 1 is the simplest way to achieve equality in ideal case.

$$T_i = \frac{Number\ of\ Players}{Number\ of\ Threads}$$

Figure 1: Ideal workload for each thread

Before we start implementing the algorithm, several utility structure and functions must be created first.

Region-Thread Mapping: To track which region is assigned to a specific thread, an array of Hash Sets is created when the world map is generated. The array is allocated with the size of number of threads, and each Hash Set holds references to regions. When a region is created, it is assigned to a thread, and this mapping relation is stored in the Hash Set for that thread.
FindMostPlayerThread(): To find the most players thread, we implemented a utility function that reads the array of Hash Sets, calculates the sum of players in each region in each thread and return the thread number with highest count.
FindBestRegion(): To find the best regions to move, another utility function is implemented that reads the Hash Set for a specific thread and an ideal number of player to move. The task for this function is iterate the hash set and find the first region with a player count less than or equal to the ideal number of players. When a region is a match, this function immediately returns.

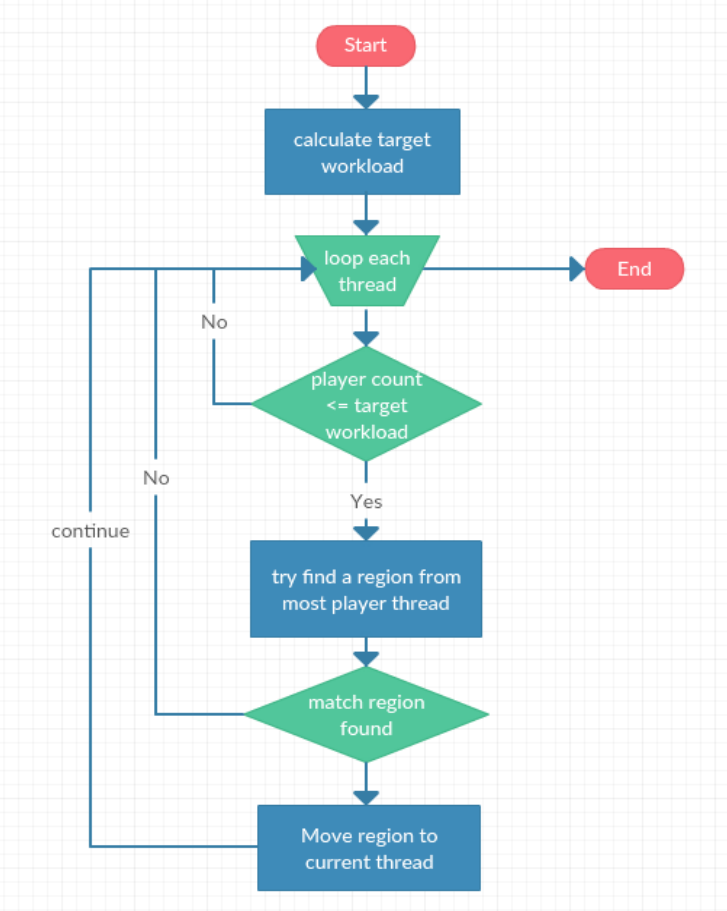

Figure 2: Spread Algorithm workflow

Finally, our spread algorithm does the following: First, a target workload value is calculated by summing all players and divided by number of threads. Then, iterate each thread in **region-thread mapping set**. For each thread with a player count lower than the target workload value, the algorithm moves a region from most player thread by calling **FindMostPlayerThread()** and **FindBestRegion()** respectively. This step is repeated until current thread player count reaches the target workload value or no more region can be moved. **Figure 2** describes the workflow for the algorithm.

4. Analysis and Evaluation

4.1. Processing Time Composition

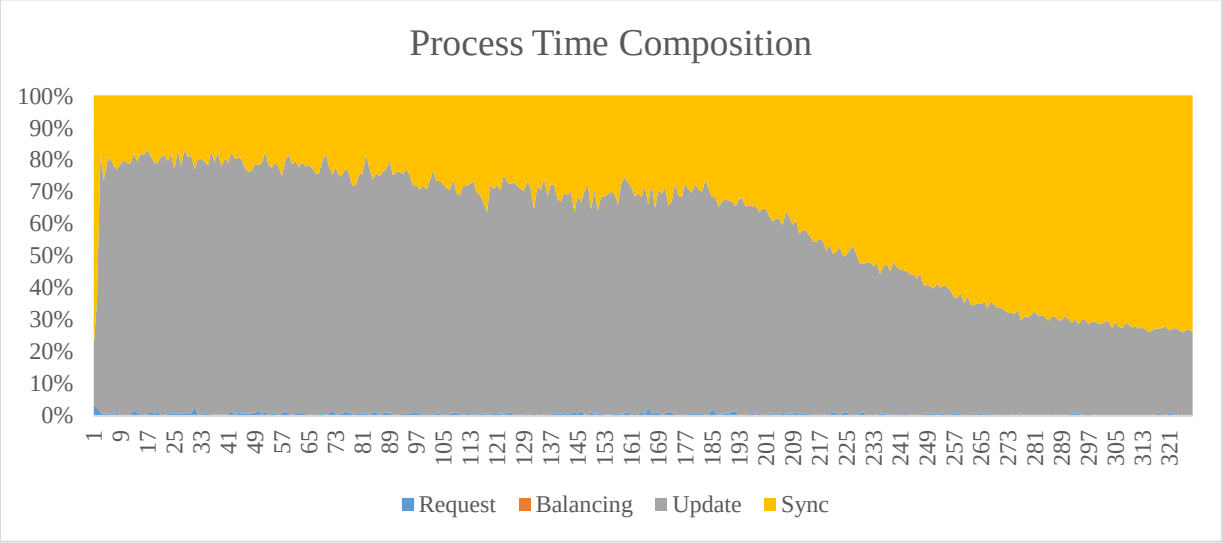

Figure 3: Process time Composition in Spread

Given the server has a multithreaded design and each thread executes 3-staged loop, we analyzed the composition of processing time. From our experiment result, both the *static* and *spread* algorithm have the similar proportion of each part. As shown in Figure 3, when no quest is present, update processing time takes the most amount of time (around 70%) while synchronization ranks second (around 29%). This trend shifts when a quest is active due to players concentrated in a single region, causing overload of corresponding thread. Only little amount of time (< 1%) has the request used, and the time used by the load balancing is nearly unnoticeable, which proves that the load balancing algorithm is efficient with little processing time.

4.2. Comparison between Static and Dynamic Loading Algorithm

4.2.1. Number of Players

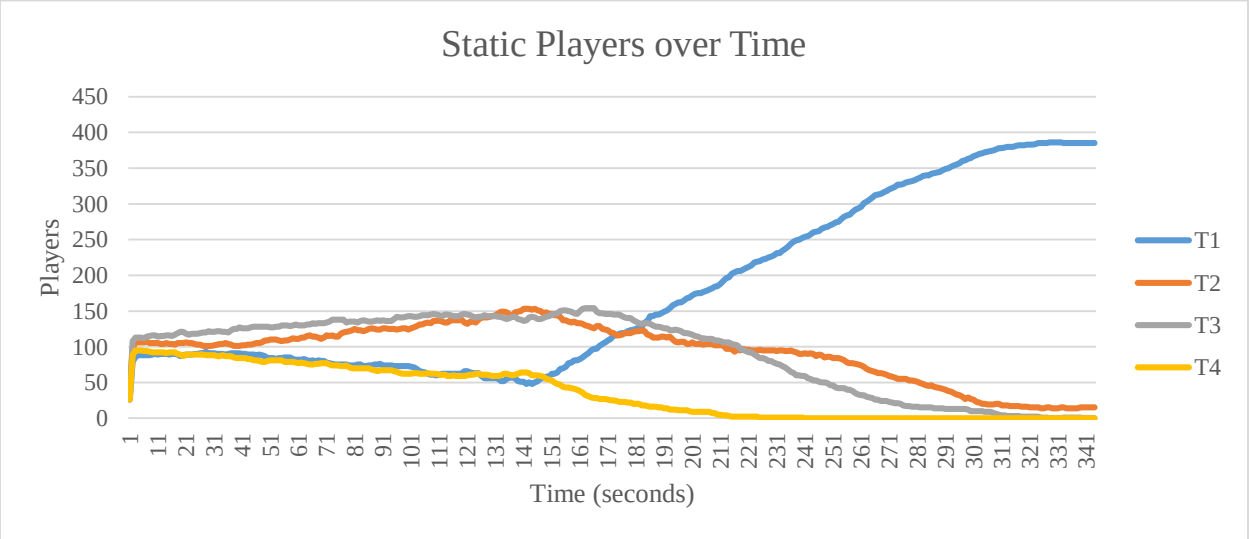

Figure 4: Players per Thread for *Static*

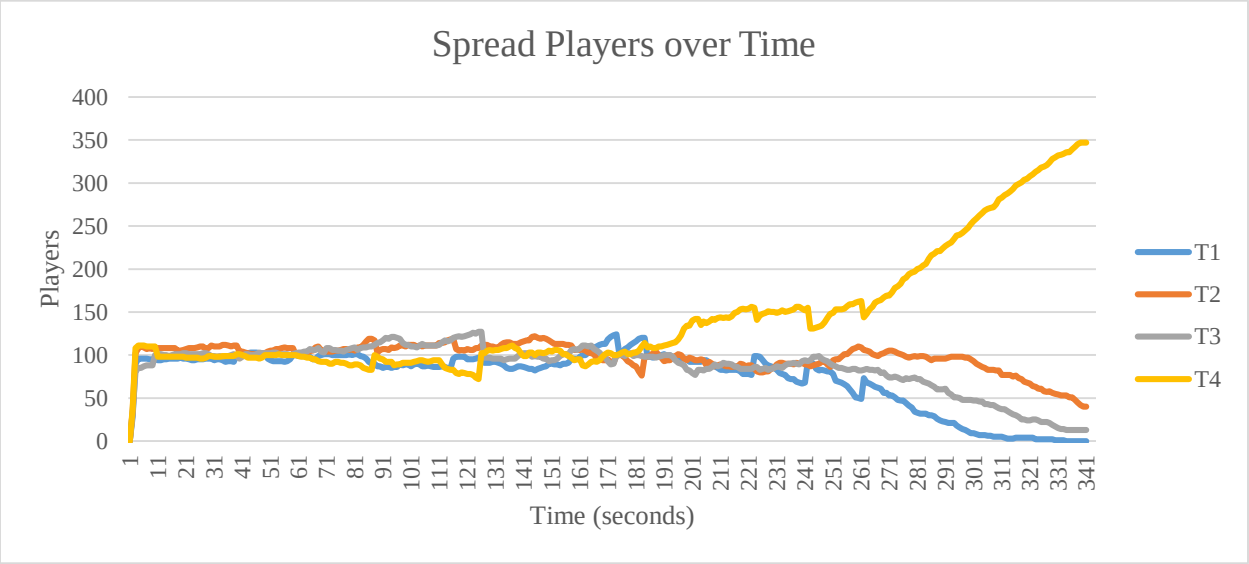

Figure 5: Players per Thread for *Spread*

Figure 4 and Figure 5 show player distribution on each thread for *Static* and *Spread*. Data shows that *Static* has uneven distribution of players, especially after 150s where almost all players are concentrated to Thread 1 at 341s. On the other hand, *Spread* has players allocated in each thread uniformly for most of the time. Unfortunately, after a certain period of time of an active quest, players eventually are gathered into a single thread. We concluded such behavior is due to players are moving to a single region which cannot be transferred to another thread. In addition, we could see that in *Spread*, players spend less time in moving to another region, which proves that *Spread* has better performance in processing the requests than *Static*.

4.2.2.Number of Requests

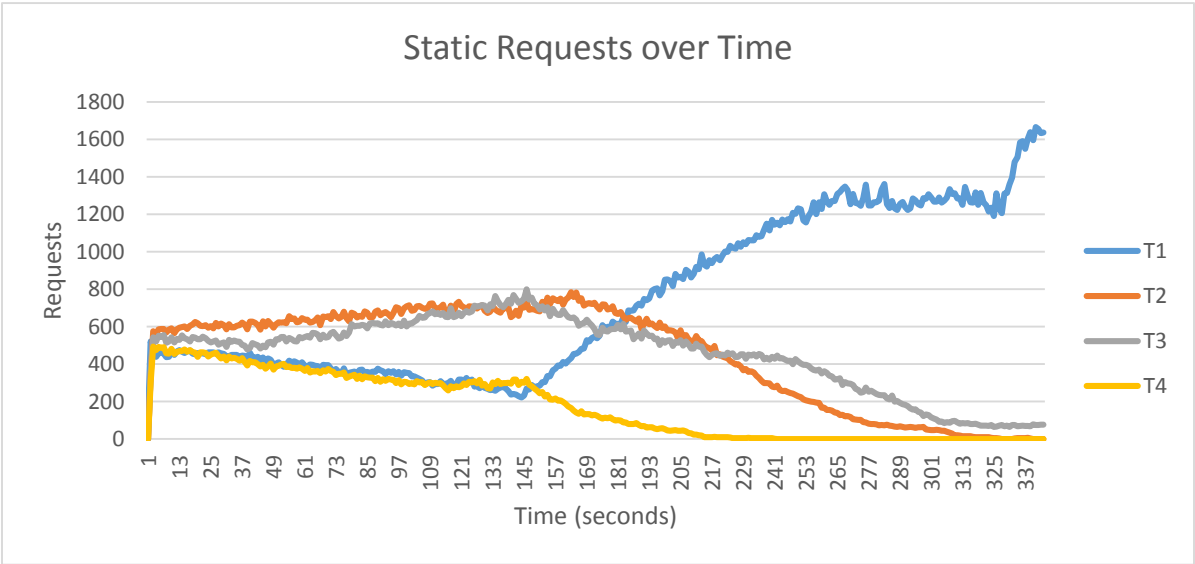

Figure 6: Requests per Thread for *Static*

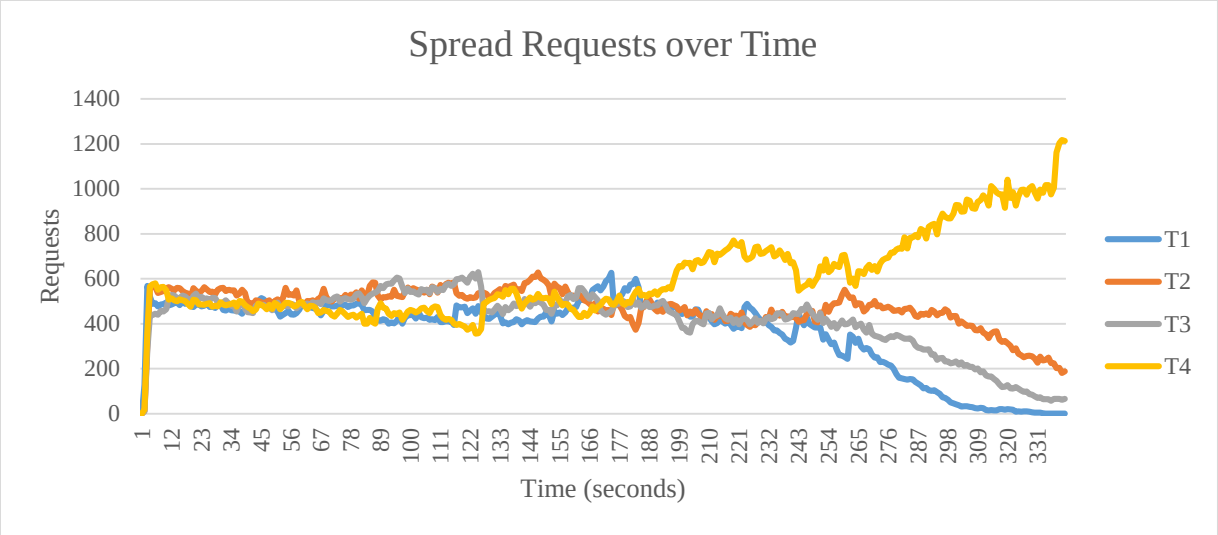

Figure 7: Requests per Thread for *Spread*

Figure 6 and 7 show the requests per second in each thread for *Static* and *Spread*. They shows a strong correlation between player distribution and request distribution. These two figures confirm the conclusion we put forward in 4.2.1 that a quest requires the players to move to another region comes in around 150s in *Static* and 260s in *Spread*. All the threads in these two schemes deal with approximately 500 requests in no quest phase and after the quests come in, larger amount of requests are sent to Thread 1 in *Static* and Thread 4 in *Spread*, which are the destinations.

4.2.3.Number of Client Updates

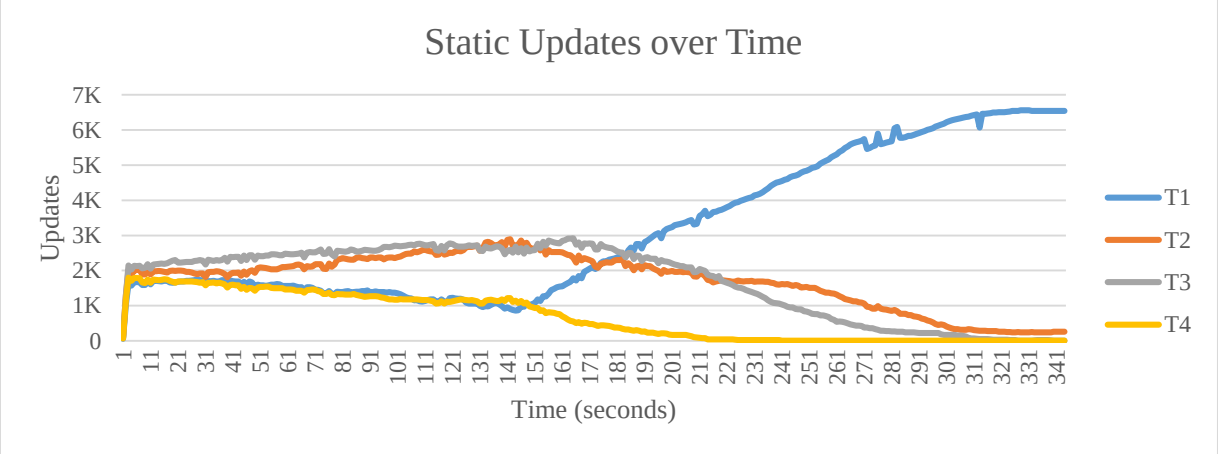

Figure 8: Client Updates per Thread for *Static*

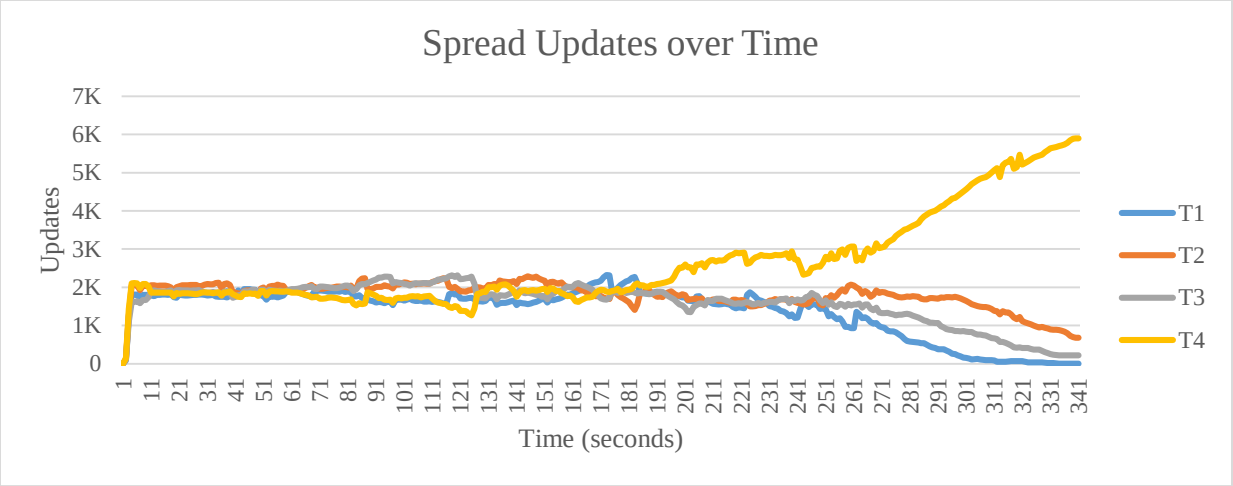

Figure 9: Client Updates per Thread for *Spread*

Figure 8 and 9 show the number of client updates per second in each thread in *Static* and *Spread*. In *Static*, more client updates are sent in Thread 1 and it reaches 6k in 150s (150s - 300s). But in *Spread*, the clients updates to Thread 4 reaches 6k in only 80s(260s - 340s). Server reaches the third stage in the 3-staged loop that each thread sends back world updates to clients in shorter time in *Spread*, which proves that *Spread* beats *Static* in processing the quests.

4.2.4.Number of Client Updates

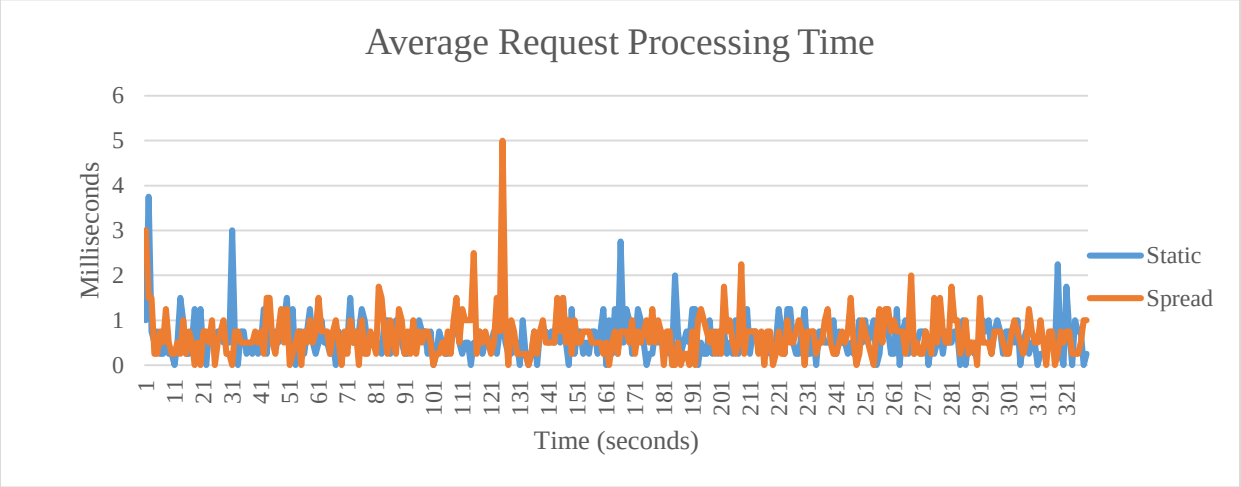

Figure 10: Comparison in Request Processing Time

Figure 10 shows the comparison in the average actual processing time of client requests between *Static* and *Spread* algorithm. Considering that this time is only a very small percentage in the entire processing time, it is reasonable that it is quite similar for those two algorithms.

4.2.5.Barrier Wait/Synchronization Time

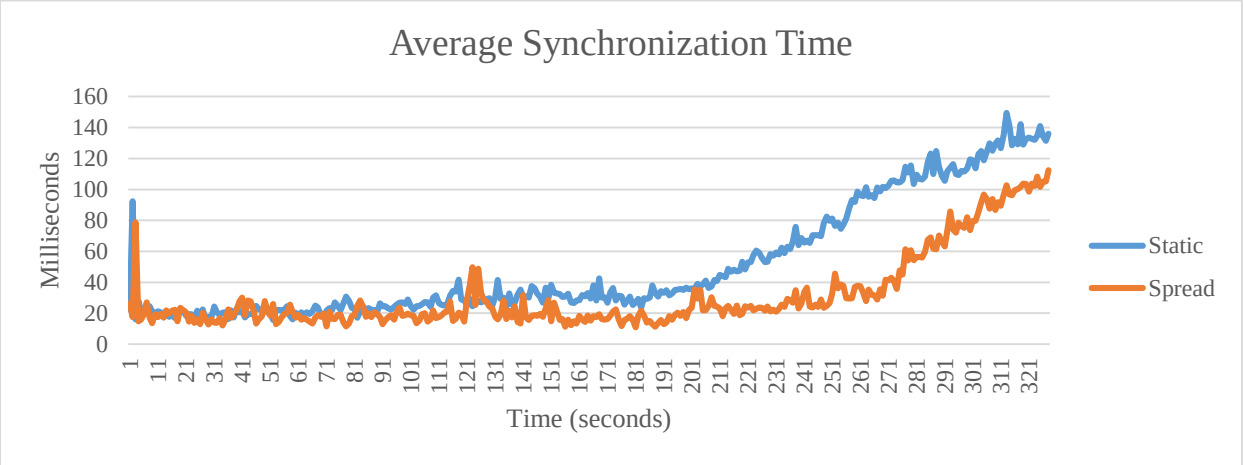

Figure 11: Comparison in Synchronization Time

As we discussed before, synchronization consumes the second longest time. In addition to both of them following similar trend, for this part, the performance of *Spread* has a slight advantage than *Static* during the no quest period (average is 30ms and 20ms respectively). However in the active quest period, synchronization time for *Spread* is drastically smaller than the time for *Static.* A close analysis reveals that *Spread* has similar workload for all threads, yielding a small barrier wait time, as oppose to *Static*, where only one thread is processing all requests.

4.2.6.Update Processing Time

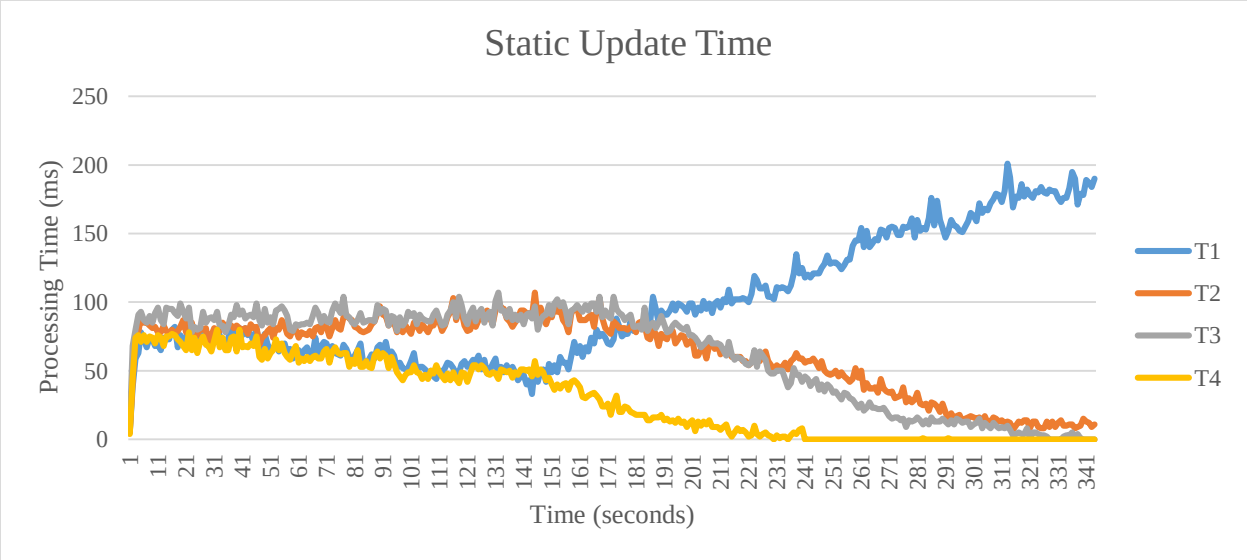

Figure 12: Update Processing Time for *Static*

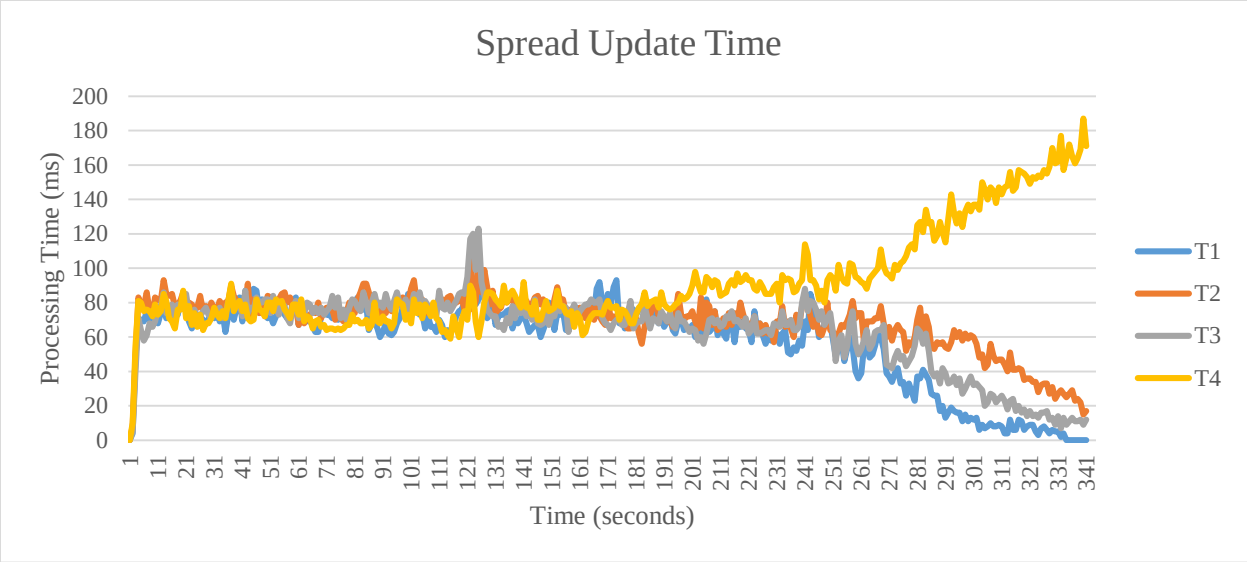

Figure 13: Update Processing Time for *Spread*

Processing client updates occupies the longest time during the whole progress. The average of *Spread* is more stable around 60-80 ms while *Static* fluctuate a lot (between 50 - 100ms) during no quest period. Even in active period, the peak time on Thread 4 for *Spread* (180 ms) is slightly shorter than *Static* (200ms), which also proves the advantage of *Spread*.

4.2.7. Composition in Load Difference

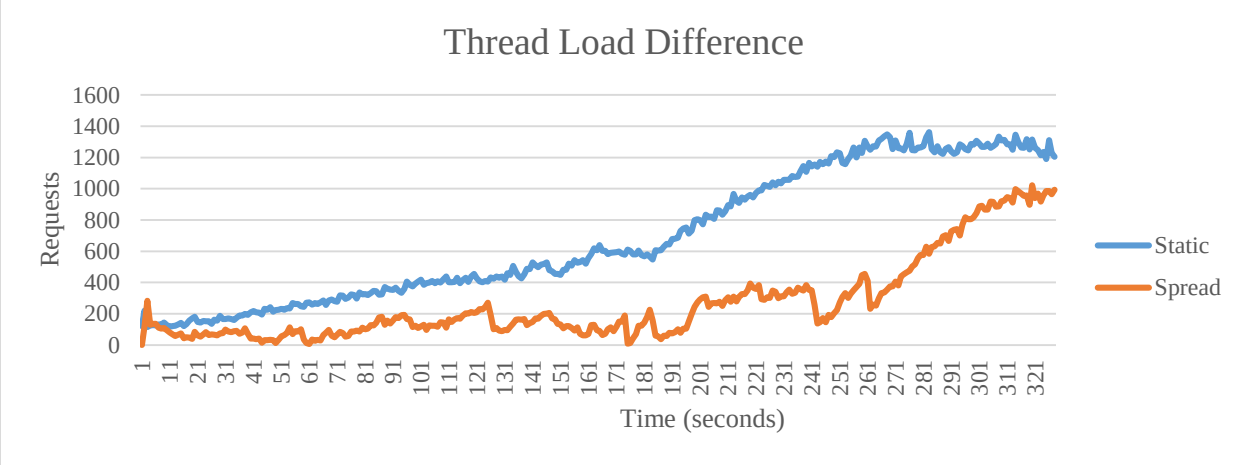

Figure 14: Process time Composition in Spread

Figure 14 is the comparison of load difference between the max-loaded thread and the min-loaded thread for *Static* and *Spread*. In both no quest and with quest period, this difference of requests of *Static* is far more than the one of *Spread*, which proves superiority of *Spread* from one side.

4.3. Comparison in region sizes for Spread Algorithm

We also explored the problem found in 4.2.1 where, even in *Spread*, players/requests/updates are heavily loaded on a single thread during active quest. As mentioned in previous section, we noticed that most players are concentrated into a single region after a certain period of time. Since a region is the smallest object we can assigned to a thread, such heavy load cannot be break down and assign to other threads.

In light of our finding, we propose that the smaller a region size is, the lower possibility of overflowing on a single thread. We have tried three different configuration for different region sizes, 8x8, 4x4, 2x2 respectively.

Figure 15 and 16 give an overview of our *Spread* algorithm's performance with 2x2 setting of region size. During no quest phase, the 2x2 configuration show similar pattern as regular 8x8 configuration: workload on each thread are equally distributed. When an active quest comes in, both the number of client requests and the number of client updates on each thread are closer to the average for a longer period of time (break at 260s), comparing with figure 7 and 9 (break at 200s) under the default 8x8 setting. Also notice that even the workload balance is broken eventually, the heaviest loaded thread in 2x2 configuration process around 600 requests and 4500 updates comparing to 1200 requests and 6000 updates in 8x8 configuration.
Finally, we acknowledged that 2x2 configuration has a bigger fluctuation than 8x8 configuration. This is very likely due to overheads for frequent region re-assignment.

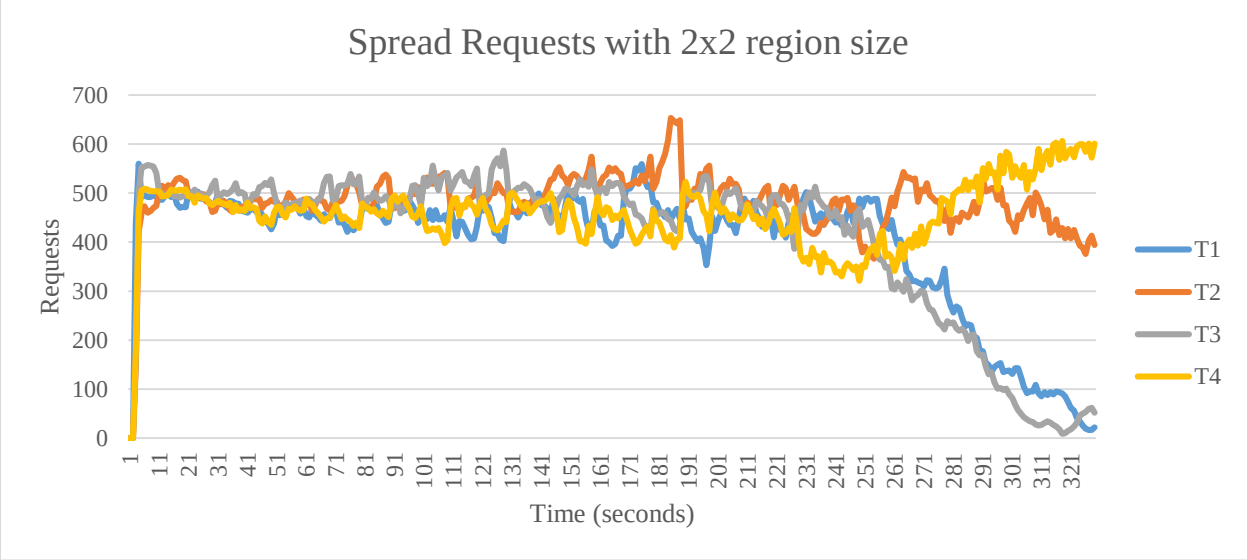

Figure 15: Request per Thread for Spread in size 2x2

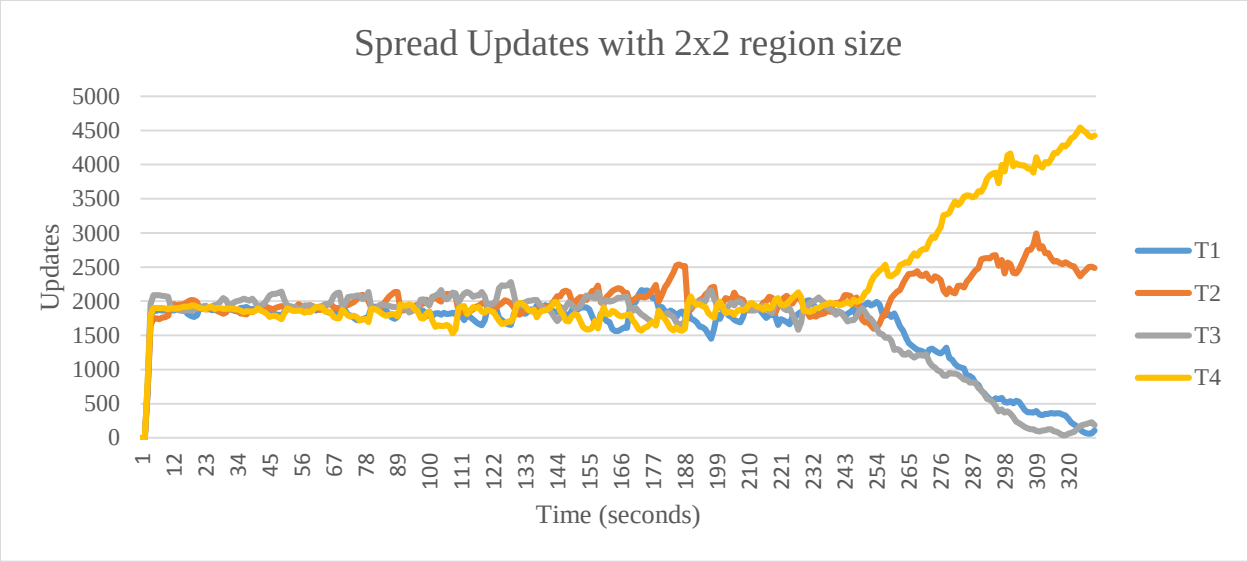

Figure 16: Replies per Thread for Spread in size 2x2

5. Conclusion

Generally speaking, the performance of *spread* algorithm outweighs *static* algorithm from many aspects. A brief summary of our conclusions is as follow:

First of all, the *Spread* algorithm processes more requests/updates than *Static* in a certain time which proves it does have better performance on saving processing time.

Second, thread/CPU utilization is far better under *Spread* algorithm than *Static* ones.

Finally, the size of region influences load-shedding a lot because a region is the smallest unit. If it is too big, it would be hard to assign it to any thread when the heavy loaded. However if the size is too small, fluctuation happens due to frequent re-assigning overheads. So wisely choosing the size of region is a pretty tricky problem.

References

[1] Chen, Jin, et al. "Locality aware dynamic load management for massively multiplayer games." *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2005.