Splitting Overlay Network for Peer-to-Peer-based Massively Multiplayer Online Games

Cheng LIU, Wentong CAI
Parallel and Distributed Computing Center
School of Computer Engineering
Nanyang Technological University
Singapore 639798
{LIUC0012, aswtcai}@ntu.edu.sg

Abstract: Massively Multiplayer Online Games (MMOGs) gradually become one of the most popular internet applications. Traditional client-server architecture is widely used in MMOGs’ deployment, but its scalability and maintenance are limited mostly by servers. Peer-to-Peer (P2P) architecture, which attempts to make use of computer resources from the computers in the network, is considered to be a candidate infrastructure for MMOGs. In this paper, we investigate some related work and propose an algorithm to split overlay network for P2P-based MMOGs. We show the benefit of splitting the overlay network in reducing lookup latency of game objects through experiments and analytical analysis.

1 Introduction

Massively Multiplayer Online Games (MMOGs) are growing rapidly with the development of computing power and network. They provide gaming environment for hundreds of thousands of players around the world. A number of MMOGs, such as [wow04], have great commercial success and attract more and more attention in related research fields.

Client-server architecture is widely employed in the current MMOG implementations. Players can access the game world by connecting to the centralized servers using their own computers. The more players one server holds, the more revenue it may generate for game vendors who maintain the servers. However, the connected players will consume the bandwidth and computing power in the server. Therefore, clusters of servers are usually used to manage game play to make MMOGs scalable. But the increase in the number of players may require more servers to be added into the clusters.

Peer-to-Peer (P2P) architecture is a new distributed computer architecture designed for sharing computer resources. It has been employed in different types of applications, such as instant messaging applications, distributed computing and file sharing. Without the necessity of a centralized server or an authority, resources distributed in this architecture can be accessed by joined peers. Because it is designed to be self-organized and has good characteristics in scalability and robustness, peer-to-peer architecture is emerging to be a
suitable architecture for distributed applications, including MMOGs.

There are hundreds of thousands of items or objects in the virtual world of MMOGs. Those objects can be accessed simultaneously by players. Two major problems in MMOG are how to inform a player the objects or players nearby in the virtual environment and how to keep game objects’ states consistent among players. In client-server architecture, servers are usually employed to collect the states of all game objects and inform the players the updated game states. However, in a P2P infrastructure, players’ computers serve as peers in the overlay network and the game objects are created and maintained by these peers. During game-play, the players need to raise queries first to obtain the game states near its position in the virtual world. The experience of the players can be improved with less network latency in game objects’ lookup. Therefore, the main objective of this paper is to reduce the lookup latency in a P2P overlay network for MMOGs.

We will present the existing related work in Section 2 and propose a split algorithm in Section 3. After that, we will demonstrate the benefit of reducing lookup latency through both experiments and analytical analysis in Section 4. We will discuss more issues in Section 5 and conclude the paper in Section 6.

2 Related Work

P2P overlay network structures can be categorized into unstructured and structured ones according to their content placement. In unstructured peer-to-peer overlay network, content can be put anywhere and the queries can be implemented through some mechanisms (e.g., flooding); whereas, the overlay is controlled delicately in structured network where contents are placed at some specified locations and the queries can be routed efficiently through distributed routing tables.

A common idea of using P2P network in online games is to make the whole big game world partitioned into multiple regions which are then assigned to peers. B. Knutson et al. proposed their MMOG architecture named SimMud [KLXH04] which was implemented on a general structured P2P overlay named Pastry [RD01] and a scalable application level multicast infrastructure (i.e., Scribe [RKCD01]). Game world is divided into several fixed rectangle regions, where players in the same region communicate with each other in a multicast group managed by Scribe. All peers and the game regions are mapped to uniformly distributed IDs in a 128-bits name space in Pastry. The peer, who has the closest ID to a game region ID, is chosen to be coordinator for that game region. Coordinators are not only responsible for message gathering and synchronization, which are related to the events happened in the game region, but also act as the root of a multicast tree for message delivery. Players in different regions must communicate with the help of the coordinators.

As it supports a $d$-dimensional virtual space for data location, the structured P2P network CAN [RFH+01] provides a straightforward way to map the partitioned game region onto servers [RWF+07] or supernodes [RMO08] which are selected from peers. Game states are managed by the server or supernode according to their locations in the game world, and every server or supernode only needs to know the $2d$ direct neighbors. With the func-
tionality (e.g., message routing, topology updating) of CAN, the regions can be merged or split dynamically. Moreover, the replication of the game states in neighbors can help in node failure recovery as well as improving of lookup performance when the player needs the information from the adjacent game regions.

Colyseus [BPS06] is a distributed architecture for multiplayer games and a modified Quake II is supported. It was implemented on a range-queriable structured overlay, called Mercury [BRS02]. Mercury creates a route hub for each attribute (e.g., different dimensions) in the application schema. Meanwhile, it organizes peers in a circular overlay while keeping adjacent peers responsible for a contiguous range of keys. Instead of region-based partitioning of game world, area of interest (AOI) filtering is implemented directly in this range-queriable overlay. Moreover, object location metadata and queries are likely to exhibit spatial locality, and thus can be mapped directly onto the overlay. This allows the players to circumvent routing paths and get the needed objects by caching recent routes.

In the traditional DHT (Distributed Hash Table) protocols such as Chord [SMLN+03], CAN [RFH+01] and Pastry [RD01], each peer in the network is assigned a unique identifier and is responsible for a certain part of key space equally. The query of keys are routed closer to the peer whose identifier most closely matches the key. Some hops may incur large network delay because peers may route the messages to a far-located peer in the underlying network. In order to overcome this problem, the topology aware lookup protocols (e.g., [RGRK04]) were proposed by considering the proximity of peers. However, the extra storage and communications are required to create the secondary lookup overlay with peers that are located closely according to the physical topology.

Some researchers focused on reducing the number of lookup hops through a large index of peers. Li et al proposed a DHT protocol called Accordion [LSMK05] in which the routing table size can be adjusted according to the rate of churn and network size. It can achieve $O(1)$ lookup latency when bandwidth is plentiful and churn is low, and $O(\log N)$ lookup latency in high churn environment and the available bandwidth is low.

Parallel lookup [LSM+05] and replication of lookup key [DLS+04] are two common but important methods to improve lookup performance for DHTs, especially under churn. In a parallel lookup, multiple lookups are initialized simultaneously by the originator. Together with iterative routing, multiple copies of query messages are sent out in each hops of parallel lookup. The whole lookup process can continue without being blocked even when some stale peers are met, so lookup retry can be avoided. In addition to improve the lookup performance, key replication can also handle the problems caused by churn. By copying the data keys to other peers, the lookup can still get the result even though the peer which is responsible for the data key leaves the system. Most of the current research is concentrated on the methods to choose the suitable peers to put a replica of the data key.

3 Our Approach

We follow the approach of DHT in MMOGs. We observe that players’ cooperation is very popular in modern massively multiplayer online role playing games (e.g. a team of
players fight together to kill a monster in the game world). Therefore, in our approach, we concentrate on players’ actions with game objects rather than obtaining information of a small area in the game world, and this motivates us to use the DHT to query game objects directly rather than game regions. As these game objects can be designed in the stage of game developing, we make an assumption that each object has a unique identifier and player are aware of the corresponding identifiers of certain objects which it wants to get. When playing the online games, players must know who are currently modifying the state of the object they are interested in and where they can get the object’s current state. The peers which can provide the information about objects are named suppliers in our approach. One important issue in online games is that the object’s state should be maintained consistently among a group of players. This is guaranteed by the game object’s supplier. The supplier receives the modification from the players and then disseminates the latest game object’s states to the players.

Rather than discussing the game objects management in DHT, we focus on a DHT split algorithm to improve the lookup of game objects. Through DHT split, the original DHT will be divided into several DHTs and the average number of lookup hops can be reduced as the query messages will be routed within a group of peers. Moreover, the network latency of game object’s lookup can be improved further if the geographical locations of peers are taken into account in determining peer groups.

4 Split Chord Ring

The DHT based on Chord [SMLN+03] is employed in several P2P applications, so we take Chord ring under low churn as the example to demonstrate our DHT split algorithm in this paper. Each peer in Chord is assigned a unique identifier using hash function. All peers have one direct successor and predecessor. They form an identifier circle, named Chord ring. Each peer has a key space ranging from its predecessor’s identifier to its own. The peer whose key space covers the hash value of key should respond to the query for such a key.

![Figure 1: Chord rings](image)
Table 1: Definition of variables for node n, using m-bit identifiers

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>finger[k]</td>
<td>first node on circle that succeeds ((n + 2^k - 1) \mod 2^m), (1 \leq k \leq m)</td>
</tr>
<tr>
<td>successor</td>
<td>the next node on the identifier circle; finger[1].node</td>
</tr>
<tr>
<td>predecessor</td>
<td>the previous node on the identifier circle</td>
</tr>
<tr>
<td>neighbor</td>
<td>one node in the other ring</td>
</tr>
</tbody>
</table>

Table 2: Definition of neighbors and split level in node n

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>neighbor[i]</td>
<td>the neighbor in i split</td>
</tr>
<tr>
<td>level</td>
<td>current split level; times of split</td>
</tr>
</tbody>
</table>

As it is shown in Figure 1, the circle represents the original Chord ring and the rest two lines represent the connection of two new rings after split. Peer n2 whose direct successor in the original ring is n3 will take n5 as its new successor after ring split. The identifier of Key 1 is covered by n6’s key space in the original Chord ring. After ring split, n6 and n7 can respond to the query of Key 1.

4.1 Single Split

The original Chord ring can be split into two small rings. In order to avoid two isolated rings, peer in the newly split Chord ring will take one peer in the other ring as its neighbor. The neighbor for each peer will be decided during the ring split. We follow the notations in Chord and add one neighbor for each peer in Table 1.

Algorithm 1 presents the procedure of splitting Chord ring. It begins with the procedure to determine whether one peer and its successor will belong to different Chord rings after split. Checking is done according to some general properties, e.g., the geographical locations of peers. The neighbor and successor in each peer must be updated during ring split. If peer n finds that its successor will belong to a different ring, it will become its successor’s neighbor and ask its successor to find a new successor for it. If n.successor and n.successor’s successor are in the same ring, they will become peers in the new ring (both of them have the same neighbor n). Otherwise, n.successor returns its successor as the new successor of n. The finger table entries will be updated periodically using the original procedure in Chord.

4.2 Multiple Split and Forwarding Messages

The original Chord ring can be split for several times. One ring will be split into two each time. The split level of a ring is used to represent how many times the split operation has been performed when the ring is generated from the original one. Two new Chord rings will have the same split level after each ring split. The original Chord ring is in split level 0, and the level will be increased by one from the former ring after splitting. Each peer
will now maintain a list of neighbors, in addition to keep track of its level (See Table 2). After each ring splitting, peers in the ring will increase its *split level* first and add one more *neighbor* to the neighbor list. To keep track of the split level and to add new neighbors to the neighbor list, the *send_message* procedure in Algorithm 1 is modified (see Algorithm 2).

With neighbors in different levels, we can forward a message in different rings wisely and avoid flooding the P2P network. There is a corresponding level for each message. When a message is generated, it is sent to every neighbor in the neighbor list. The level of the message is set according to the level of its neighbor in the list. We describe the message forwarding in Algorithm 3.

**Algorithm 1** Pseudocode of splitting a Chord ring

```plaintext
Procedure: n.split()

inSameR = check_in_same_ring(n, successor)

if ! inSameR then
    successor.remove_predecessor()
    successor.send_message(NEIGHBOR, n)
end if

Procedure: n.send_message(msg_id, n')

inSameR = check_in_same_ring(n, successor)

if msg_id is NEIGHBOR then
    neighbor = n'
    if inSameR then
        successor.send_message(NEIGHBOR, n')
    else
        n'.send_message(SUCCESSOR, successor)
    end if
else if msg_id is SUCCESSOR then
    successor = n'; n'.notify(n)
end if

Procedure: n.notify(n')

if predecessor is nil or n' ∈ (predecessor, n) then
    predecessor = n'
end if
```

**Algorithm 2** Pseudocode of adding new neighbor

```plaintext
Procedure: n.send_message(msg_id, n')

inSameR = check_in_same_ring(n, successor); level = level + 1

if msg_id is NEIGHBOR then
    neighbor[level] = n'
    if inSameR then
        successor.send_message(NEIGHBOR, n')
    else
        n'.send_message(SUCCESSOR, successor)
    end if
else if msg_id is SUCCESSOR then
    successor = n'; n'.notify(n)
end if
```

**Algorithm 3** Pseudocode of message forwarding

```plaintext
Procedure: n.remove_predecessor()

if predecessor is not nil then
    predecessor = nil
end if

Procedure: n.fix_fingers()

next = next + 1

if next > m then
    next = 1
end if

finger[next] = find_successor(n + 2

Procedure: n.find_successor(id)

if id ∈ (n, successor) then
    return successor
else
    return successor.find_successor(id)
end if
```
Algorithm 3 Pseudocode of forwarding messages in rings

Procedure: n.send_to_neighbors(message)

for i ≤ level do
    if message.level < i then message.level = i; neighbor[i].send_to_neighbors(message)
end if
end for

5 Analysis and Experiments

In this section, we evaluate the network latency of lookup after we split a Chord ring according to the peers’ geographical locations. Whether two peers belong to the same ring or not can be decided by using landmarks approach [ZZZ+04]. We assume that the function check_in_same_ring in Algorithms 1 and 2 is defined and the network latency of the peers in the same group is less than the latency of peers in different peer groups. A key-value pair which is held by one peer in the original Chord ring can be copied to the peers in the other rings after split according to the key’s popularity in different peer groups. Here we consider two simple polices:

- **Without replication:** After ring split, the key-value pair is still in the same peer as in the original ring.

- **With replication:** There is one copy of the key-value pair in a peer for each ring. Therefore, peers can get the query result from a peer in the same group.

5.1 Performance Analysis

We first consider the difference of total number of hops after and before ring split without the mechanism of replication. Suppose that a ring (r) is split into two rings r1 and r2 with m1 and m2 peers respectively. For the queries raised by peers in ring r, the total number of hops changed, δ1, can be calculated by:

\[ H_{after} = \frac{m_1^2}{2} \log_2 m_1 + m_1 \times m_2 \times (1 + \frac{\log_2 m_2}{2}) + \frac{m_2^2}{2} \log_2 m_2 + m_2 \times m_1 \times (1 + \frac{\log_2 m_1}{2}) \] (1)

\[ H_{before} = \frac{(m_1 + m_2)^2}{2} \log_2 (m_1 + m_2) \] (2)

\[ \delta_1 = H_{after} - H_{before} \] (3)

\( H_{after} \) represents the total number of hops after the ring r is split. After r is split into two, a peer in either ring r1 or r2 can start key lookup. The destination peer which responds to the query could belong to any one of these two rings. Four terms in equation (1) represent four different combinations of originator-destination pairs. If the originator and destination peers are in the same ring (e.g., ring r1), the number of hops is \((\log_2 m_1)/2\) according to the result in [SMLN+03]. But, if they belong to different rings (e.g., the originator in ring r1 and the destination in ring r2), the number of hops in each lookup should include one hop through the neighbor link and the number of hops to route the query in the other ring.
Assume that there are \( n \) peers that are not in ring \( r \). If they raise a key lookup for the key located in a peer in ring \( r \), the difference in the total number of hops, \( \delta_2 \), is calculated as follows. There are two possibilities to get the key for the peer outside ring \( r \) – if its neighbor and the destination peer are in the same ring (e.g., ring \( r_1 \)), only \( \log_2 m_1/2 \) hops is needed; else one extra hop is included to forward the query to the ring where the destination peer belongs (i.e., ring \( r_2 \) in this case) in addition to the number of hops required to route the query in ring \( r_2 \). Therefore, the total number of hops can be calculated by:

\[
n * m_1 \ast (\frac{m_1}{m_1 + m_2}) \ast (\frac{\log_2 m_1}{2}) + n * m_1 \ast (\frac{m_2}{m_1 + m_2}) \ast (1 + \frac{\log_2 m_1}{2})
\]

This can be simplified as:

\[
n * m_1 \ast (\frac{m_2}{m_1 + m_2} \ast 1) + n * m_1 \ast \frac{\log_2 m_1}{2}
\]

The same reasoning can also be applied when the neighbor is in ring \( r_2 \). Hence, we have:

\[
H'_{after} = n * m_1 \ast (\frac{m_2}{m_1 + m_2} \ast 1) + \frac{n * m_1}{2} \log_2 m_1 + n * m_2 \ast (\frac{m_1}{m_1 + m_2} \ast 1) + \frac{n * m_2}{2} \log_2 m_2
\]  \( \quad (4) \)

\[
H'_{before} = \frac{n * (m_1 + m_2)}{2} \log_2 (m_1 + m_2)
\]  \( \quad (5) \)

\[
\delta_2 = H'_{after} - H'_{before}
\]  \( \quad (6) \)

So, from equations (3) and (6), the total number of hops changed by ring \( r \)’s split is:

\[
\delta_H = \delta_1 + \delta_2
\]  \( \quad (7) \)

When a peer in the \( n \) peers that are not in ring \( r \) tries to find a key located in one of these peers, the number of hops of these queries will not be affected by the ring split. The number of hops of queries is also not considered in the above analysis when a peer in \( r \) tries to find a key located in a peer that are not in ring \( r \).

Then, we analyze the improvement of network latency for key lookups after ring split. Assuming that \( L_1 \) and \( L_2 \) stand for the average network latency for each pair of peers in rings \( r_1 \) and \( r_2 \) respectively, we can get the average network latency of \( r \) as

\[
L_{avg} = \frac{m_1^2 \ast L_1 + m_2^2 \ast L_2 + 2 \ast m_1 \ast m_2 \ast L_o}{(m_1 + m_2)^2}
\]  \( \quad (8) \)

Where \( L_o \) represents the average network latency of the neighbor links.

Following the similar analysis above, we add the network latency to equations (1), (2), (4) and (5). So, the change of lookup latency, \( \delta_L \), can be calculated as follows:

\[
L_{after} = \frac{m_1^2 L_1}{2} \log_2 m_1 + \frac{m_2^2 L_2}{2} \log_2 (m_2) + \frac{m_1^2 L_2}{2} \log_2 (m_2) + \frac{m_2^2 L_1}{2} \log_2 (m_1)
\]

\[
L_{before} = \frac{(m_1 + m_2)^2}{2} \log_2 (m_1 + m_2)
\]

\[
L'_{after} = n \ast m_1 \ast (\frac{L_o}{m_1 + m_2}) + \frac{n \ast m_1 \ast L_1}{2} \log_2 m_1 + n \ast m_2 \ast (\frac{L_o}{m_1 + m_2}) + \frac{n \ast m_2 \ast L_2}{2} \log_2 m_2
\]
\[ L'_{before} = \frac{n \times (m_1 + m_2) \times L_{avg}}{2} \log_2 (m_1 + m_2) \]

\[ \delta_L = L_{after} - L_{before} + L'_{after} - L'_{before} \] (9)

We also analyze the number of hops and lookup delay changed after ring split when replication mechanism is used. Since keys are replicated, in this case a query can always be answered by a peer in the same ring. In the following analysis, only queries raised from peers in ring \( r \) are considered. When a peer in a ring raises a key lookup, the corresponding average number of hops is \((\log_2 m_i)/2\) where \( m_i \) is the number of peers in the new Chord ring. The difference of the total number of lookup hops after and before ring \( r \)'s split, \( \delta_H \), can be calculated using equation (10), where \( m_1 \) and \( m_2 \) are the number of peers in rings \( r_1 \) and \( r_2 \) respectively after split and \( m \) is the number of peers in the original ring (i.e., \( m = m_1 + m_2 \)).

\[ \delta_H = \frac{m_1 \times m \times \log_2 m_1}{2} + \frac{m_2 \times m \times \log_2 m_2}{2} - \frac{(m_1 + m_2) \times m}{2} \log_2 (m_1 + m_2) \] (10)

We follow the definition of the average network latency in equation (8) and get the change of the lookup latency after and before ring split as follows:

\[ \delta_L = \frac{m_1 \times m \times L_1}{2} \log_2 m_1 + \frac{m_2 \times m \times L_2}{2} \log_2 m_2 - \frac{(m_1 + m_2) \times m \times L_{avg}}{2} \log_2 (m_1 + m_2) \] (11)

5.2 Simulation

We first build a small game world with 200 game objects. The underlying network topology with 300 nodes is generated by BRITE [MLMB01] using Waxman model. The size of main plane is set to 1000 and the propagation delay between two direct-connected nodes is less than 5 time units. The shortest route between any two nodes is calculated using Dijkstra’s algorithm. 300 players are mapped randomly to the nodes in the network topology. All players move randomly and perform 600 times of game object lookups in the following scenarios:

**Scenario 0:** the original Chord ring with 300 peers before split

**Scenario 1:** two Chord rings with 50 and 250 peers after split from the original one

**Scenario 2:** three rings with 50, 104 and 146 peers after the ring with 250 peers is split

The network latency for object lookup is collected and the distribution is shown in Figure 2. 68% lookup are less than 35 time units in scenario 1, and 74% lookup are less than 35 time units in scenario 2 without object replication. But there are only 52% of them before split. With the object replication, 85% and 97.5% lookup which are less than 35 time units can be identified in scenario 1 and scenario 2 respectively.

We use BRITE again to generate a larger topology using the same setting. 9947 players are mapped randomly to the nodes in the network topology and these nodes form a Chord ring. Then the original Chord ring is gradually split into 2, 4, 7, 8 and 10 rings and the peers in each ring are determined by their geographical locations. After each split, we move those 9947 players randomly to query an object out of 15,000 objects in the game world. The average number of hops \( (H_i) \) and the network latency \( (L_i) \) for 12000 times
of game object lookups are recorded. In order to check whether the lookup performance is improved when the peer geographical location is considered, we normalize the results with the following functions where \( H_{\text{original}} \) and \( L_{\text{original}} \) stand for the average number of hops and lookup latency in the original Chord ring respectively:

\[
E_{\text{hop}} = \frac{H_i}{H_{\text{original}}} \quad E_{\text{latency}} = \frac{L_i}{L_{\text{original}}}
\]

We also evaluate the number of hops and lookup latency according to our analysis, and compare the result to the experiments. The number of hops and lookup latency after splitting several times comparing to the numbers in the original Chord ring is calculated by (12), where \( \delta_H \) and \( \delta_L \) are defined in equations (7) & (9) and (10) & (11). In order to simplify the computation, we suppose that \( L \) is the average latency between peers in the same ring (so both \( L_1 \) and \( L_2 \) are equal to \( L \)), and \( L_o = c \cdot L \) \((c > 1.0)\) is the average latency between neighbors in the different rings. There are \( m \) peers in the original ring with the average network delay among peers is \( L_{\text{chord}} \).

\[
F_{\text{hop}} = 1 + \frac{\sum \delta_H}{(m^2 \cdot \log_2 m)/2} \quad F_{\text{latency}} = 1 + \frac{\sum \delta_L}{(m^2 \cdot L_{\text{chord}} \cdot \log_2 m)/2}
\]

The comparison of our experimental result and analytical result is shown in Figure 3 with \( c = 1.015 \). As shown in the figure, the experimental and analytical results are very close to each other. The average number of hops and the lookup latency is reduced gradually. So we can use the analytical formulas to analyze the performance of large P2P networks or to analyze performance under various network conditions. For example, the result of the network latency improvement with different values of \( c \) is shown in Figure 4. It demonstrates that the bigger the difference of the network latency between and within peer groups, the more significant improvement the ring splitting can achieve.

Figure 2: The distribution of lookup latency without/with replication
6 Conclusions

In this paper, we presented an approach to make use of structured P2P network for massively multiplayer online games. Chord is applied to locate the game objects maintained by the suppliers. We also proposed an algorithm to split Chord to achieve better lookup performance. We applied the algorithm to reduce lookup network latency by grouping peers that are geographically close to each other. Our evaluations in both analytical and experimental aspects demonstrate the benefit of splitting.

References


