

On Mitigating Network Partitioning in Peer-to-Peer Massively Multiplayer Games

Yuan He^{1,2}, Yi Zhang^{1,2}, and Jiang Guo³

¹ Laboratory for Internet Software Technologies, Institute of Software,
The Chinese Academy of Sciences, Beijing 100080, China

² Graduate School of the Chinese Academy of Sciences, Beijing 100039, China
{heyuan, zhangyi}@itechs.iscas.ac.cn

³ Department of Electrical and Computer Engineering,
University of Toronto, Ontario, Canada
jguo@eecg.toronto.edu

Abstract. Recently, peer-to-peer infrastructure has been proposed to support massively multiplayer games in the literature. However, when underlying network partitions due to network outages, the game world will partition into several parallel game worlds and it is difficult and costly to merge them when the network partitions disappear. Existing approaches resort to a centralized server to arbitrate. Aiming at mitigating the effects brought by network partitions, we propose a fully distributed algorithm based on state-stack matching. Our theoretical analysis and numerical results show that our approach can resolve the merging issue at the least loss of game states with high probability.

1 Introduction

Massively multi-player games (MMGs) have a long history following a centralized infrastructure. Recently, researchers are proposed to use Peer-to-Peer overlays to support massively multi-player games (MMGs) [1]. In such a peer-to-peer gaming infrastructure, except that a central server is required to keep player account information, all other game states are stored in a distributed way all over all peers participating in the game.

In most MMGs, the player assumes the role of a character in a virtual world. A typical multiplayer game world is made up of immutable terrain, characters controlled by players (PCs), mutable objects such as food, tools, weapons, and non-player characters. Different game states have different access patterns and consistency requirements.

Existing P2P approach [1] proposes to store object states in a distributed way. Furthermore, copies of object states are replicated in order to increase availability with the presence of node failures or network outages. However, such a P2P approach leads to an undesired problem: in case of network outages, the underlying network partitions; the system can continue to allow shared states access, but with no communication between partitions, the original game world splits into two or more parallel worlds, likely with loss of consistency. Things go worse with potential paradoxes when the network condition recovers and partitions are about to merge.

As a rule in a peer-to-peer massively multiplayer game, consistency is most important: we can not imagine that a game full of paradoxes has the possibility to attract players. We start from existing P2P approach [1] and focus our attention on mitigating the adverse effect caused by network partitioning or the consistency issue. Simply put, in this paper, we explore two questions:

- 1) Can we keep the consistency of split game worlds?
- 2) When merging partitioned game worlds, can we solve conflicts with least game states loss?

The remainder of this paper is organized as follows. In Section 2 we briefly introduce the existing approach using P2P overlay support MMGs. In Section 3 we present our proposed merging algorithms based on state-stack matching and theoretical analysis. Section 4 shows experimental results using numerical method. Section 5 concludes this paper.

2 P2P Support for MMGs

Some methods have been worked out [1], [2], [3] and work well under the assumption of low failure frequency and graceful network behavior. However, performance becomes poor in the face of network partitions as introduced in Section 1. Without effective communication or coordination during the course, paradoxes probably can't be avoided.

In order to ensure availability and consistency in the face of network partitions and merge, we propose a distributed strategy of dynamic state management. It is based on the coordinator-based mechanism proposed in [1], which will be briefly introduced as follows.

2.1 The Coordinator-Based Mechanism

Building the whole system on top of the classical Pastry peer-to-peer infrastructure, we can group players and objects by regions according to their game situations (Fig. 1), and distribute the regions onto different peers by mapping them to the Pastry key space. For example, players in a same game scene have closest relationship. These peers, together with all those objects in this scene, form their common region. Each region is assigned an ID, computed by hashing the region's textual name using a collision resistant hash function. [2], [4]

Each region or object is assigned a coordinator, to which all updates information are sent. The coordinator both resolves conflicting updates, and is a repository for the current value of the corresponding object. In a comparatively common design, the coordinator not only coordinates all shared objects in the region, but also serves as the root of the multicast tree, as well as the distribution server for the region map. [5] Although mapping all synchronization and update information to a single node simplifies the system design, it might incur a high network load on this coordinator if the game complexity increases. However, the load can be distributed by assigning a different ID for each type of object in the region, thus mapping them onto different peers.

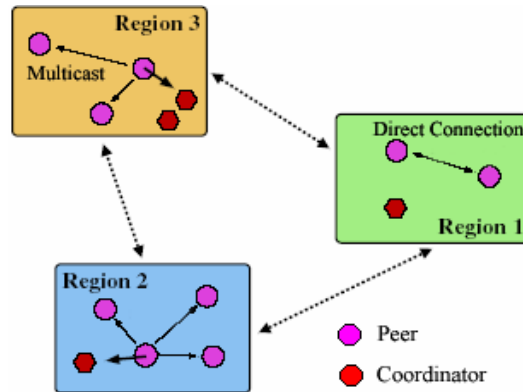


Fig. 1. Regions, peers, and coordinators

Successful updates are multicast in periodic message forms in the scope of current region to keep each player's local copy fresh. To keep consistency among players, timely message delivery is necessary, which will be discussed in part B. A primary-backup mechanism [1] has been applied to tolerate fail-stop failures of the network. Node Failures are detected using regular game events, without any additional network traffic. The coordinator's state is replicated once a failure is detected. The algorithm tries to keep at least one replica up under all circumstances, to prevent losses. As our objective is not replication algorithm, but the strategy to deal with network partitions and merge, we just give a simple introduction of replication algorithm here, and don't make any more comments on this issue in the following sections.

3 Mitigating Network Partitioning

When the scenario of network partitions appears, the original game world is partitioned into several parallel worlds. Then we try to force each parallel game world to be independent normal world. When the partitions again merge, states possessed by different parts are combined in a smart way, potential paradoxes are removed and then valid game state still can be attained.

3.1 Independent Game Worlds Rebuilding on Network Partitions

In our distributed strategy, we embed in a periodic message mechanism to ensure consistency and availability. Messages between coordinators and common peers are sorted to three classes:

Message a: Request for current local state on peers, sent to all common peers by the coordinator.

Message b: Combined updated state of whole region, produced by the coordinator after it gets each peer's latest local state (message c), and then multicast to all peers.

Message c: Peer's current Local state, sent by a common peer to the coordinator.

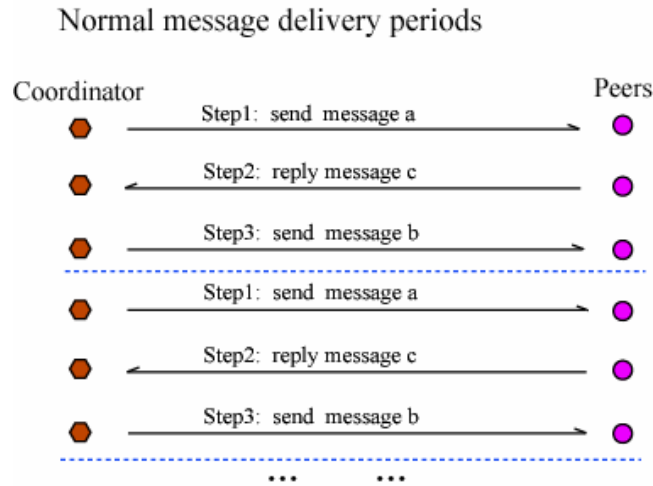


Fig. 2. Periodic messages delivery

Generally, the coordinator first multicasts message *a* to all peers in the region. Receiving message *a*, each peer send its current local state copy in format of message *c* to the coordinator as response. After getting all messages *c*, the coordinator removes the conflicts and produces an updated region state. Then the coordinator multicasts message *b* to all peers with the content of region state update. This is called a normal message delivery period, shown in Fig. 2. We can set the coordinator's request frequency at a proper fixed value according to the network conditions and average node abilities so as to meet the demands of a peer-to-peer game.

Now we continue to discuss the system's behavior when network partitions happen. Without loss of generality, we just discuss the scenario with only a single coordinator in each region and assume that an original region is just partitioned into two parts. As shown in Fig. 3, region *O* is partitioned into *A* and *B*. Peers in region *A* and *B* lose the communication with peers in the other region.

We suppose node *P* is the coordinator. Immediately after partitions, peers in region *B* can not receive message *a* from *P*. Therefore, *P* can not receive response from peers in region *B* either. At this moment, *P* is able to ascertain happening of partitions. It can also make sure which peers are still connective, and which peers are not.

Then *P* should adjust the state possessed by it: First, it marks those connections and information related to the peers without response invalid, which means those peers are not in the same region as *P* now. Second, *P* generates a new state update only with support of the messages from connective peers. Because *A* and *B* are new regions, they may be assigned a new region ID computed by hashing. As a result, *A* and *B* will both have new coordinators of their own (P_a , P_b). Suppose *P* remain in *A*, then *P* should send its newly produced region state update to P_a and leave the job to P_a to multicasts message *a* inside region *A* when the next message delivery period starts.

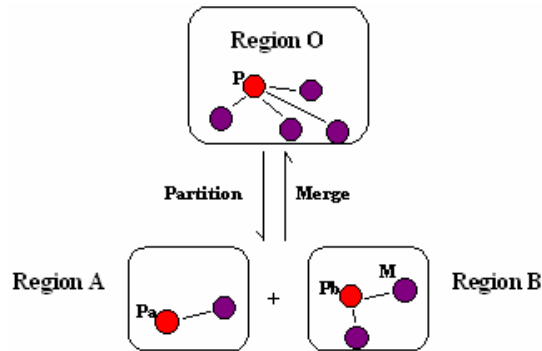


Fig. 3. Region partitions

On the aspect of other peers, peers in region B (such as M) can not receive any messages from their former coordinator P. After this state keeps over the length of a message delivery period, each peer is able to ascertain happening of partitions. Connections between these peers and P_b will be set up. P_b will take the job as the new coordinator of region B and immediately starts a new message delivery period.

In this way, region A and B with new coordinators are properly rebuilt after network partitions. Because invalid peer states and information have been removed during rebuilding, the new coordinators (P_a , P_b) only keep the necessary state of their own region and no conflicts exist now. Independent game worlds come into being and the game successfully continues.

3.2 Regions Merge Mechanism

Through the mechanism in part B, parallel game worlds are rebuilt after network partitions and peers in each region behave without conflicts. Yet, these regions are about a same unique game scene in fact. Each takes itself as the only representative of this scene in the whole game world. Accesses and changes to an object perhaps take place concurrently, so their independent behavior will probably cause paradoxes when they try to merge again, which threaten consistency.

In [1], a central server blessed mechanism is adopted to keep consistency. But it assumes that partitions still keep connections with a central account server. If network partitions thoroughly, including outages to the central server, coordination among regions can't be ensured without the central server and paradoxes after merge can't be avoided yet.

In our distributed strategy, the central server is no longer needed. Coordinators are strengthened to keep consistency and reduce the conflict probability, designed as follows: Each coordinator not only possesses the latest region state, but also keeps a state-stack, which stores a certain number of recent region states, from stack bottom to its top in time order. (Fig. 4) Once the coordinator produces a new region state update, it pushes the state into the state-stack while it multicasts the state to other peers. Considering the game's complexity, the low conflict probability and the average peer capability, we may set the state-stack to a reasonable size N .

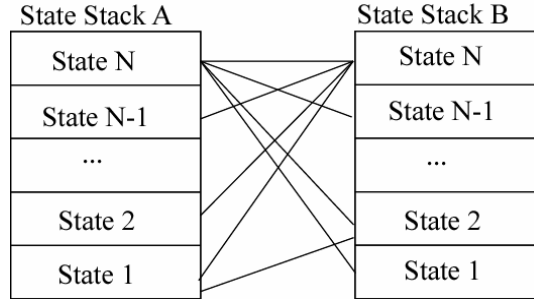


Fig. 4. State-stacks and matches of states

On the inverted condition of Fig. 3, let's focus on the scenario when regions merge. Without loss of generality, we assume each region's coordinator has already kept full state-stack storing slots of states. When merge attempt starts, two coordinators of region and compare their stored states in pairs. Generally each state is compared to the states in the other stack. A pair of states without any conflicts is called good match. At most N^2 times of compares will be executed and we'll eventually get set of good matches. One match will be selected and combined into reasonable state of the merged region O. This match of states without any conflicts between each other ensures that there isn't any paradox in region O.

Later states match before combination equals to less loss of state information after merge. For example, we have two matches, (9, 8), (6, 9). Match (9, 8) will be selected, because the latest state sequence number is (10, 10), and match (9, 8) loses only periods of states in all, comparatively less than periods lost by match (6, 9). Unless the good matches-set is empty, at least 1 match will be found. Successful combined state is then sent to the new assigned coordinator of region O. Subsequently, normal message delivery period in region will start. Merging region A, to region succeeds then.

Even if the good matches set is empty at first, things keep unchanged for while. When next message delivery period begins, new state will be pushed into each stack. Then we go on to compare the new state with states stored in stack before. If any good matches are found, regions can be merged immediately. Otherwise, we may still keep on this circulatory process to seek good matches. Through the relevant experiments in Section 4, we can see that fairly high probability of successful merges can be obtained. And at the same time, the most states information can be reserved.

3.3 Theoretical Analysis of Network Partitions Process

3.3.1 Time Complexity

A complete process of resolving network partitions is composed of two segments:

$$T = T_1 + T_2$$

T: whole process time;

T_1 : time for coordinator to ascertain network partitions;

T_2 : a message delivery period for the coordinator and peers to set up connections and rebuild new region state.

We assume the time of a normal message delivery period is T_0 .

T_2 will never start until the coordinator ascertains the partitions in T_1 . In our distributed strategy, the only rule for the coordinator to judge partitions is that it has not received any response message from some certain peers after the time for all peers to send back messages c passes, as is indicated by Fig. 2. Then T_1 equals to the interval between the network partitions and the receipt of messages c in one period. Considering the unpredictable network faults might appear any time in the period at a completely same possibility, we can draw a conclusion as follows:

$T_1 = 0.5 * T_0$ averagely.

On the other hand, the length of T_2 obviously equals to T_0 .

$$T = T_1 + T_2 = 0.5 * T_0 + T_0 = 1.5 * T_0 \quad (1)$$

T_0 is assigned a proper fixed value according to the network conditions and average node abilities to meet the demands of a P2P game. From (1) we can see that, once the coordinator's request frequency is set, the complete process time of resolving network partitions is fixed, no matter how many peers exist in the network or how they partition. Usually T_0 is only a very short interval (at most 1 second level), so our strategy gives an effective solution against network partitions in P2P games.

3.3.2 Overhead

During the process of partitions, messages are sent between peers and coordinators for game worlds rebuilding. To simplify the following computation, let's assume messages a , b , and c have same length and M messages should be delivered for resolving problems during an X -peers region partitions. Now we compute proportion M/X , the average individual overhead.

Without loss of generality, a region is also assumed to be partitioned into two parts with X_1 and X_2 peers each. $X = X_1 + X_2$. From discussion in the previous parts, we've seen that all messages to be delivered during network partitions are the messages sent in a whole normal message delivery period. One more special message should be counted in, which is the state-copy sent from the former coordinator to the new coordinator in one region (such as P and P_a in Fig.3). Then,

$$M = 3 * (X_1 - 1) + 3 * (X_2 - 1) + 1 = 3(X_1 + X_2) - 5 = 3X - 5$$

$$M/X = 3 - 5/X \quad (\text{if } X \rightarrow \infty, M/X = 3)$$

In the real peer-to-peer networks, number of peers in a region is usually very large. We draw a conclusion that the average overhead equals to a light level of 3 messages delivery per peer during the whole process of partitions.

4 Simulations

We develop simulation program according to the region merge mechanism proposed above. In practical P2P game, it has big states set with large number of different region states. However, not all states conflict with each other. We can group them by classes under the rule that conflicts don't appear inside class and exist between any two classes inevitably. If two states are in the same class, they will form good match and then can be combined. As for two regions which need to merge, if at least one good match exists in the state-stacks of their respective coordinators, merge will

certainly succeed. In order to simulate the stochastic distribution of the game states, states in all slots are generated stochastically. 1000 times of merge tests will be performed and the probability of successful merges can be eventually obtained.

Related parameters, N : stack size, K : number of state classes, P : probability of successful merges.

4.1 Experiment 1

We set $K=1000$, use increasing values for N , and check P 's variety, as shown in Table 1 and Fig. 5.

Table 1 and Fig. 5 come up with the following conclusion: For certain P2P game with fixed number of states, larger state-stacks on coordinators lead to higher probability of successful merges. In our design, $K=1000$, $N=50$ effectively achieve high probability of successful merges over 91%.

Table 1. Probability as $K=1000$

N	K	P
5	1000	0.025
10	1000	0.095
15	1000	0.203
20	1000	0.328
25	1000	0.465
30	1000	0.595
40	1000	0.797
50	1000	0.919

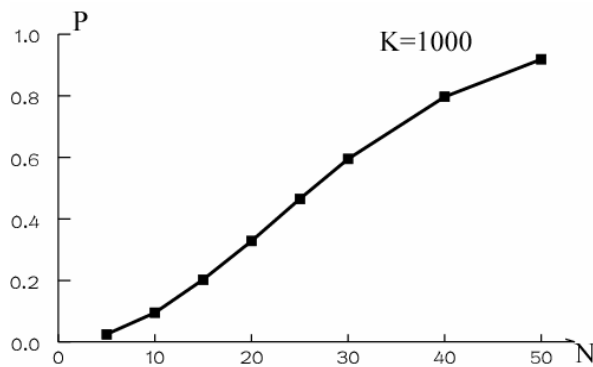


Fig. 5. Probability variety as $K=1000$

4.2 Experiment 2

What size of state-stack is necessary for coordinator if we want high success probability? Satisfactory to everybody, farther results have been found through times of experiments.

Table 2 shows only part of our experimental results. From these results, we can see that when N^2/K equals to a constant, P is relatively localized around a fixed value too. The detailed theoretic analysis is too complicated and we may just simply consider that N^2/K entirely determines the value of P .

Based on the data listed, rules may be formed as follows: If we want probability P no less than about 90%, N^2/K should equal to at least 1. That's to say, $N = \sqrt{K}$; If we want higher probability no less than about 98%, N^2/K should equal to at least 4. That's to say, $N = 2\sqrt{K}$.

Table 2. Probability as $K=100, 400, 900, 1600$

N	K	N^2/K	P
10	100	1	0.632
15	100	2.25	0.895
20	100	4	0.981
20	400	1	0.633
30	400	2.25	0.895
40	400	4	0.982
30	900	1	0.633
45	900	2.25	0.896
60	900	4	0.982
40	1600	1	0.633
60	1600	2.25	0.896
80	1600	4	0.983

As N is the size of a coordinator's state-stack. From the discussions above, we can draw a conclusion: For a certain degree of success probability, space complexity of the region merge mechanism is $O(\sqrt{K})$.

On the other hand, each slot in the state-stacks usually describes its corresponding game-state with a marked bit-map structure, the variation of each bit representing one entity in the game scenes. Accordingly the memory cost of each slot is about 1KB on normal occasions and not more than 10KB at most.

Therefore, for the duty of keep essential state-stack in the region merge mechanism, the memory cost of each coordinator is: $C = 10KB * N = 20\sqrt{K}$ KB at most. Even if the number of state classes K increases fairly large, e.g. $K=1*10^6$, C still keeps at a relatively economic level of 20 MB, which may be easily provided by all the participant PCs.

5 Conclusion

Network faults such as network partitions and merge have deep negative impact on availability and consistency. On top of Pastry infrastructure and the coordinator-based mechanism, our distributed strategy makes outstanding contributions on resolving network partitions and merge.

Embedded in periodic message mechanism, independent parallel game worlds are rebuilt after network partitions in an effective way without any additional spending. By introducing the state-stack structure to the coordinator-based mechanism, the game system in our design gets the ability in the face of regions merge. Under the rule of choosing latest state matches for combination, the most game-states can be reserved after merge. Proved by the analysis and simulations results in the end, our strategy runs with good time and space efficiency.

Comparing with other P2P game systems such as the central server based mechanism introduced in [1], systems with our distributed strategy has the ability in the face of thorough network partitions and merge. However, more work should be done for better states replication algorithms. Security in the particular process of network partitions and merge is another field we are going to research in the future.

Acknowledgment

This work is supported by the National Natural Science Foundation of China under Grant No.60373053; One-Hundred-People Program of Chinese Academy of Sciences under Grant No.BCK35873.

References

1. B. Knutsson, H. Lu, W. Xu, and B. Hopkins: Peer-to-Peer Support for Massively Multiplayer Games. In IEEE Infocom (March, 2004).
2. A. Rowstron and P. Druschel: Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001) (November, 2001).
3. E. Cronin, B. Filstrup, and A. Kurc: A Distributed Multiplayer Game Server System. In University of Michigan technical report (May 2001).
4. E. Sit and R. Morris: Security Considerations for Peer-to-Peer Distributed Hash Tables. In First International Workshop on Peer-to-Peer Systems (IPTPS '02) (March, 2002).
5. H. Lu, B. Knutsson, M. Delap, J. Fiore, and B. Wu: The Design of Synchronization Mechanisms for Peer-to-Peer Massively Multiplayer Games. In Penn CIS Tech Report (2004).