INTERACTIVE MULTI-SERVER ONLINE GAMING

USING PARALLEL AND DISTRIBUTED COMPUTING

ABSTRACT:

The terminologies, *Parallel and Distributed Computing* have always been associated with different meanings and connotations, since time immemorial. The very basic concept of online gaming has undergone a paradigm change in the last decade.

From the days of *snake and ladder to ludo* games in the mono-syllable text phone era and the days of *temple runs and talking Toms*, the era of online gaming has undergone a boundless metamorphosis**.** *PUBG and FORTNITE* have now been the most widely practiced gaming zones that today's youngsters are clung to. The ever-changing human mindset and the ever-changing adolescent attitude drive the computing mindset so frenzy, always looking for novelties and innovations in the field of online gaming.

INTRODUCTION:

Parallel computing denotes solving an application by dividing it into processes that run simultaneously (on multiple processors). **Distributed computing** indicates solving an application by dividing it into processes that run (possibly together with other applications) on different autonomous computers that are interconnected with each other and cooperate, thereby sharing resources. The main difference lies in the criteria used to decide if an application is parallel or distributed.

This concept of computing these days has played a vital role in network-applied online games. **Networked games** are rapidly evolving from small 4-8 person, one-time play games to large-scale games involving thousands of participants [4] and persistent game worlds.

The purpose of this manuscript is to highlight the practical applications of parallel and distributed computing in the field of much sought-after online gaming. Here, we discuss the design, implementation, and evaluation of **Colyseus**, a distributed computing architecture for interactive multiplayer online games.

CONTEMPORARY GAME DESIGN:

Colyseus enables games to efficiently use widely distributed servers to support a large community of users. We have integrated our implementation of Colyseus with Quake II [8] (a popular server-based FPS game), and also have used measurements of Quake III game-play to develop our own **Colyseus**-based game and associated automated players that mimic the Quake III workload.

Contemporary game design requires a thorough determination of the requirements of multi-player games. Here**,** each player (game participant) controls one or a more avatars (player's representative in the game) in a game world (a two or three dimensional space where the game is played).

This pattern of description applies to many popular games and genres, including first person shooters (FPSs) (such as Quake, Unreal Tournament, and Counter Strike), role playing games (RPGs) (such as Everquest, Final Fantasy Online, and World of Warcraft), and others. In addition, this model is similar to that of military simulators and virtual collaborative environments.

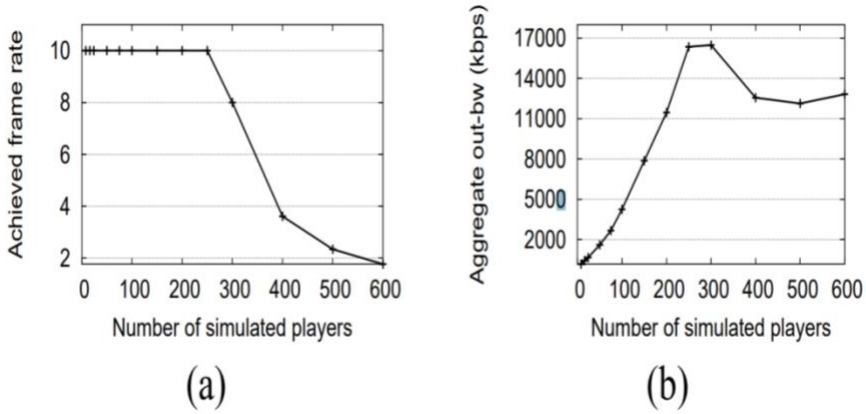
CLIENT-SERVER SCALING PROPERTIES

Scalability Analysis:

The three game parameters that most impact network performance are:

- the number of objects in play (NumObjs),
- the average size of those objects (ObjSize), and

- the game's frame-rate (UpdateFreq).



Computational and network load scaling behavior at the server end of client-server system.

Here, we only consider objects representing players (which tend to dominate the game update traffic), NumObjs ranges from 8 to 64, ObjSize is ~200 bytes, and UpdateFreq is 10 updates per second.

A naIve server implementation which simply broadcasts the updates of all objects to all game clients (NumClients) would incur an outbound bandwidth cost of NumClients × NumObjs × ObjSize × UpdateFreq, or 1-66Mbps in the case of Quake II games between 8 and 64 players.

MULTIPLAYER GAMING WORKLOADS:

WORKLOAD & EVALUATION:

In our experiments, game players are simulated with computer controlled bots8 and get respawned (re-generated) after being killed. Although, we tried several different rule sets, they did not produce qualitative or quantitative differences in our results. Each experiment ran for approximately 15 minutes and, unless otherwise specified, our results examine 5 minutes during the middle of each game. We ran experiments under both the federated and peer-to-peer deployment.

For our experiments, we designed a large world map that supports up to 200-400 players (compared to 16-64 players on most maps). Areas-of-interest cover, on average, 1% of the map. Figure 5 shows the cumulative distribution of the time that objects (excluding missiles) remained in a player's area of interest. We estimated these using replica lifetimes. Most replicas live only for a few seconds, as is expected since players move rapidly from one region to another when not fighting. Hence, the dynamic nature of this workload is likely to stress a distributed game implementation.

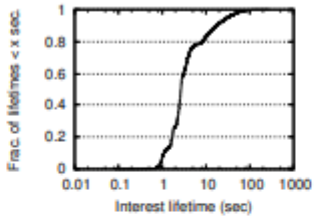
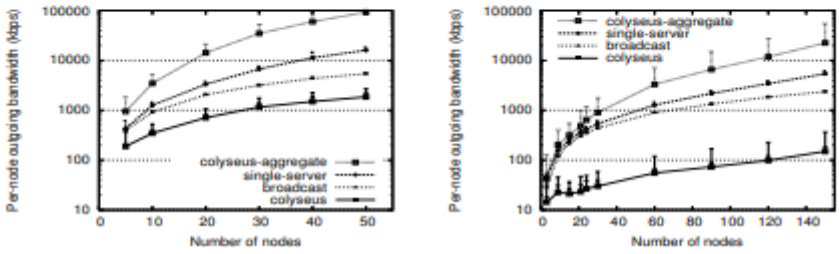


Figure 5: Cumulative distribution of interest lifetimes.



(a) Federated-servers scenario (b) Peer-to-peer scenario

Scaling in per-node bandwidth usage as the number of servers in the system increase in : (a) the federated-servers scenario and (b) the peer-to-peer scenario.

MAIN PROTOCOLS:

1 ST PROTOCOL – MIMAZE PROTOCOL

MiMaze is a totally distributed game (no server) that uses an unreliable communication system. This uses multicast to avoid a common problem of cheat. But it is not sufficient to render the real-time properties of the game with network delays and large numbers of participants.

- **Interaction delay:** any action issued by any participant must reach other participants before 100 ms.

- **Large number of participants:** The targeted number of participants is 10.000. That means existing group management policies are not applicable. For MiMaze, we have decided to limit the number of participants in order to analyze scalability problems to smaller groups. Interactive data: Actions are short (few bytes) and frequent (but not periodic). They differ from other multimedia data (audio and video).
- **High level of dynamicity in group structure and topology**: This can be amplified by the participation of mobile units.
- **Adaptation techniques**: Can be applied.
- **Data encoding:** Designed in such a way to enable hierarchical transmission of the game information.

2 ND PROTOCOL - LOCKSTEP PROTOCOL

The lockstep protocol is a partial solution to the look-ahead cheating problem in peer-to-peer architecture multiplayer games, in which a cheating client delays his own actions to await the messages of other players.

To avoid this method of cheating, the lockstep protocol requires each player to first announce a "commitment"

- **Drawback:** As all players must wait for all commitments to arrive before sending their actions, the game progresses as slowly as the player with the highest latency.
- **Overcome:** asynchronous lockstep protocol. Lockstep is a major advance in communication protocols for distributed games because it is provably secure against several cheats. Unfortunately, time progresses in the lockstep protocol at the speed of the slowest player.

PEER-TO-PEER ASPECTS:

We divide the architecture into four main components: Authentication, Communications, Storage, and Computation.

- The authentication component is responsible for controlling access to the game.
- The communication component determines how players send messages to each other.
- The storage component provides long-term storage of the world state.

AUTHENTICATION

The challenging problems with distributed authentication are security and scalability We use a hash table: that provides a unique ID WITH NAME AND expiration date To help with scalability, once the pairs are generated, they are stored on a DHT by the player or server. The advantage of using a DHT is that as the number of players increases, the storage capacity increases.

- The computation component schedules computations across the player base.

COMMUNICATION

- ***A DISTRIBUTED COMMUNICATION ARCHITECTURE*** has the possibility of supporting millions of players.
- ***CLIENT/SERVER ARCHITECTURES***, all communications must be sent to the server before being sent to players, introducing an additional delay. Distributed communication allows players to send messages directly to each other, creating a more responsive game.
- ***CONSISTENCY*** is required if players are from different regions if an error arises they won't be able to play together.
- ***DIVIDING COMMUNICATION BETWEEN ZONES***: As the players inc. we divide the zones into a virtual world to peer-to-peer groups.
- ***TO ORGANIZE THE PEER-TO-PEER NETWORKS*** into a hierarchy of groups, we use distributed election protocols to elect one or more leaders to act as nodes in the hierarchy. The nodes assist in event propagation and group location. To help build

the hierarchy of groups, we propose using a DHT to map unique zone IDs to the list of nodes in a given zone.

- ***TO PROVIDE CONSISTENCY*** and event orders using a new protocol called NEO. NEO ensures low latency and also prevents cheating.

STORAGE

• TWO TYPES:

1. **Ephemeral Data**

   • Data which is ephemeral is simply stored on the DHT, indexed by its geographical area in the virtual world. Players that enter an area, query the DHT for all ephemeral data.

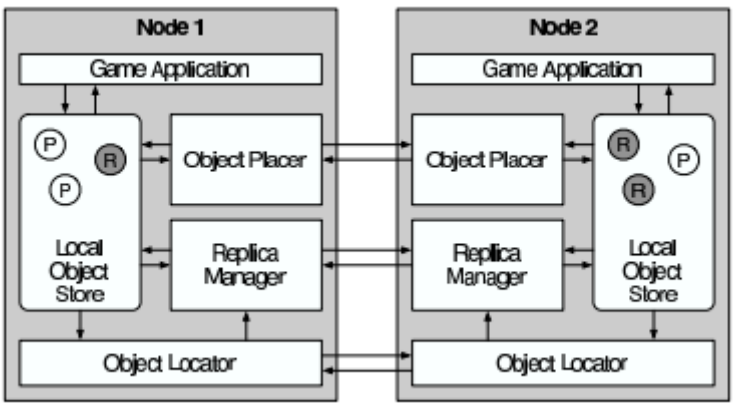
2. **Permanent Data**

   • This allows the player to make sure the data is always available when they are online, but provides a backup in case their local data becomes corrupted.

COMPUTATION

The server in a client/server architecture typically does not have enough power to simulate complex interactions between players and the artificial intelligence (AI) behind monsters and other non-player characters.

COLYSEUS ARCHITECTURE:



THE COMPONENTS OF COLYSEUS

- **Denotation:** Circled R's represent secondary replicas, circled P's represent primary objects. The figure shows the interaction between each component, the game, and components on other nodes. The primary shared interface is the local object store. The game application creates and deletes primaries in the store and periodically executes each primary's think function.
- To locate relevant remote objects, we use a robust, distributed object location component based on a publish-subscribe system called Mercury (described in Section 4). Mercury allows nodes to register interests using range queries over multiple object attributes. Our implementation shows that this is sufficient for Quake II and we believe this is rich enough to support many game genres.
- **Object Location**: Given a set of primary objects, a node requires a mechanism to discover all other objects the primaries are interested in. Replica Management:
- **Synchronization:** Once a node has determined the remote objects it requires (i.e., objects for which it does not maintain the primary), it must maintain loosely-synchronized replicas of those objects.
- **Object Placement:** Colyseus should (re)assign primary objects to nodes to optimize some metrics such as communication cost.
- Colyseus decomposes solutions to these problems into three corresponding components: an Object Locator, a Replica Manager and an Object Placer.

CHALLENGING PROBLEMS THAT MUST BE OVERCOME:

• Authenticating players and granting access rights to the game

• Maintaining consistency, ordering events, and propagating events to intended recipients

• Providing tamper-resistant storage of characters and game state

• Scheduling computations across the players

• Providing responsiveness to interactive elements of the game

• Ensuring the architecture is cheat-proof by preventing players from taking advantage of the architecture itself in order to cheat.

CONCLUSION:

In this, we have described the design, implementation, and evaluation of **Colyseus, a distributed systems' architecture** for online multiplayer games. Colyseus takes advantage of a game's ability to tolerate inconsistent states in its partitioning of states across system nodes. It also takes advantage of game software's predictable read/to write workloads to aggressively pre-fetch objects to a system node. We found that the following design choices are critical for supporting low-latency (< 100ms) game-play:

(a) decoupling object discovery and replica synchronization,

(b) proactive replication for short-lived and rapidly moving objects, and

(c) pre-fetching of relevant objects using interest prediction.

Based on our investigation, we believe a range-query DHT is a better choice as the routing substrate as compared to a normal DHT, predominantly due to its load-balancing properties.