A Fully Distributed Architecture for Massively Multiplayer Online Games

Chris GauthierDickey, Daniel Zappala and Virginia Lo
University of Oregon
Department of Computer Science
1202 University of Oregon
Eugene, OR 97403-1202
{chrisg | zappala | lo}@cs.uoregon.edu

Categories and Subject Descriptors: C.2.4 [Distributed Systems]:Distributed applications

General Terms: Design, Security, Reliability

Keywords: distributed, architecture, multiplayer, interactive, games

Historically, massively multi-player online games (MMOs) have used a client/server architecture. This architecture has the advantage that a single authority orders events, resolves conflicts in the simulation, acts as a central repository for data, and is easy to secure. On the other hand, it has the disadvantage of increased latency, localized congestion at the server, limited storage capacity, and limited computational power by the server.

We are developing a fully distributed, peer-to-peer architecture for MMOs that has the potential to overcome these problems. Players can send messages directly to each other, thereby reducing delay and eliminating localized congestion. The storage capacity created by combining the storage resources of all players can easily exceed that of a single server. Furthermore, the ability to execute thousands of remote processes in parallel on user machines provides unparalleled computational power for MMOs. Finally, this architecture would allow individuals to start their own MMO without the incredible investment in resources required by client/server architectures.

However, a distributed architecture has several fundamental and challenging problems that must be overcome. First, it must authenticate and grant access rights to players in the game. It must maintain consistency, order events, and propagate events to intended recipients. The architecture needs to provide tamper-resistant storage of characters and game state and it needs to schedule computations across the players. Last, it must provide responsiveness and be thoroughly cheat-proof.

We divide our 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 and the storage component provides long-term storage of world state. Last, the computation component schedules computations across the player base.

The authentication component is responsible for providing access control to the game. In order to authenticate securely, we assume the authentication component (AC) is trusted. The AC works by digitally signing two pairs (player_id, expiration_date) and (player_id, public_key). These pairs are published on a distributed hash table (DHT) accessible by all players. Using the DHT, players can validate other players and discover their public key without needing to contact the AC. Further, the AC can publish a special pair, (player_id, *banned*), which bans a player from the game.

The communication component (CC) is responsible for exchanging messages and events between players. The primary duties of the CC is to enforce an ordering of events between players, to keep latency as low as possible, and to prevent cheating. Our CC uses the NEO protocol [1] for small groups to provide low-latency, cheat-proof event ordering. Each NEO group then uses several leaders to form a hierarchy of groups so that events with a scope larger than the group can be propagated accordingly through the hierarchy.

The storage component (SC) provides long-term, persistent data storage. We divide storage into two categories: permanent and ephemeral. Permanent data, which must always be available, is kept safe by both the player owning the data and replicated by an underlying DHT. Ephemeral data, or data which is only temporary, is simply stored on a DHT, and recreated by the system as needed.

The computation component schedules game computations among the players. For example, monster AI could be scheduled for players to execute. While a certain amount of computations can be done locally by characters interacting with their environment, the computation component must ensure that players are not cheating. Thus, remote processes must be scheduled among players and results must be verified. We use randomly selected witnesses to perform key calculations and to ensure a cheat-proof environment.

1. REFERENCES

[1] C. GauthierDickey, D. Zappala, V. Lo, and J. Marr. Low Latency and Cheat-proof Event Ordering for Peer-to-Peer Games. In *ACM NOSSDAV*, 2004.

SIGCOMM'04 Workshops, Aug. 30+Sept. 3, 2004, Portland, Oregon, USA.
ACM 1-58113-942-X/04/0008.