A Distributed Architecture for Online Multiplayer Games

Ashwin Bharambe Jeffrey Pang Srinivasan Seshan
{ashu+,jeffpang+,srini+}@cs.cmu.edu
Carnegie Mellon University

ABSTRACT

This paper presents the design, implementation, and evaluation of *Colyseus*, a distributed architecture for interactive multiplayer games. Colyseus takes advantage of a game's tolerance for weakly consistent state and predictable workload to meet the tight latency constraints of game-play and maintain scalable communication costs. The system uses single-copy consistency and weakly consistent replication to support low-overhead read/write access to data at any node. Colyseus also provides a rich distributed query interface and effective pre-fetching subsystem to help locate and replicate objects before they are accessed at a node. We have implemented Colyseus and modified a popular first person shooter game called Quake II to use Colyseus. Our measurements of Quake-II and our own Colyseus-based game with hundreds of players shows that Colyseus is effective at distributing game traffic and computation across the participating nodes. Our results show that Colyseus can support low-latency game-play for orders of magnitude more players than existing single server designs, while requiring similar per-node bandwidth.

1 INTRODUCTION

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. However, like most Internet applications, current networked games are centralized. Players send control messages to a central server and the server sends (relevant) state updates to all active players. This design suffers from the well known robustness and scalability problems of single server designs. For example, high update rates prevent even well provisioned servers from supporting more than several tens of players for first person shooter (FPS) games. Further, client-server game designs often force players to rely on infrastructure provided by the game manufacturers. These infrastructures are sometimes not well provisioned or long-lived; thus, they either provide poor performance or prevent users from playing their game long after their purchase.

A distributed design can potentially address the above shortcomings. However, architecting a distributed application is difficult due to the challenges of partitioning the application's state (e.g., the game world state) and execution (e.g., the logic to simulate player and game AI actions) among the participating nodes. Distributing a networked game is made even more difficult by the performance demands of real-time game-play. In addition, since the game-play of an individual player translates to updates to the shared state of the game application, there is much more write traffic and write-sharing than most distributed applications.

Fortunately, there are two fundamental properties of games that we can take advantage of in addressing these challenges. First, games tolerate weak consistency in the application state. For example, even current client-server implementations minimize interactive response time by presenting a weakly consistent view of the game world to players. Second, game-play is usually governed by a strict set of rules that make the reads/writes of the shared state highly predictable. For example, most reads and writes at a node occur upon objects which are physically close to the player(s) associated with that node. The challenge, then, is to arrive at a scalable and efficient state and logic partitioning that enables reasonably consistent, low-latency game-play. This paper presents the design, implementation and evaluation of Colyseus, a novel distributed architecture for interactive multiplayer games designed to achieve the above goals.

In Colyseus, any node may create read-only replicas of any game object. However, objects in Colyseus follow a single-copy consistency model – i.e., all writes/updates to an object are serialized through exactly one primary copy in the system. Although replicas are only kept weakly consistent with the primary copy, they enable the low-latency read access needed to keep the game execution timely. This mirrors the consistency model of the existing client-server architecture, albeit on a per-object basis. An important challenge with this design is that each node must be able to determine the set of object replicas it needs in advance of executing any game logic. Colyseus relies on the predictability of data access patterns to identify key properties about the needed set of objects. Colyseus also provides a rich query interface over the system-wide collection of objects to identify and fetch these objects. We have implemented this query interface on both a randomized distributed hash table (DHT) [34] and a dynamically load balanced, range-based DHT [13]. However, even lookups on efficient DHTs can be too slow for network games. To hide this lookup latency, Colyseus uses locality and predictability in the movement patterns of players to speculatively pre-fetch objects. In addition, Colyseus only uses this lookup system to discover relevant objects. The propagation of updates between primary copies and replicas is handled using direct connections. We show that the combination of all these techniques is critical to enabling interactive game-play.

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. These concrete case studies illustrate the practicality of using our architecture to distribute existing game implementations. Our measurement of the Quake II and model game prototypes on an Emulab testbed with hundreds of players shows that Colyseus is effective at distributing game traffic and workload across the participating nodes. In addition, we show that Colyseus is able to provide each server/player with low-latency and a consistent view of the game world. In the following sections, we provide background about general game design as well as the design and evaluation of Colyseus.

2 BACKGROUND

In this section, we survey the requirements of online multiplayer games and demonstrate the fundamental limitations of existing client-server implementations. In addition, we provide evidence that resources exist for distributed deployments of multiplayer games. This motivates our exploration of distributed architectures for such games.

2.1 Contemporary Game Design

To determine the requirements of multiplayer games, we studied the source code of a few popular and publicly released online game engines, such as Quake II [8] and the Torque Networking Library [9]. In these games, 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 description applies to many popular 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 [24, 25].

Almost all commercial games of this type are based on a client-server architecture, where a single server maintains the state of the game world.[1] The game state is typically structured as a collection of objects, each of which represents a part of the game world, such as the game world's terrain, players' avatars, computer controlled players (i.e., *bots*), items (e.g., health-packs), and projectiles. Each object is associated with a piece of code called a *think function* that determines the actions of the object. Typical think functions examine and update the state of both the associated object and other objects in the game. For example, a monster may determine his move by examining the surrounding terrain and the position of nearby players. The game state and execution is composed from the combination of these objects and associated think functions.

[1]Some large scale RPGs use multiple servers. They partition the world into disjoint regions or simulate parallel, disjoint universes on multiple servers. Hence, they remain similar to the client-server architecture and are still centralized.

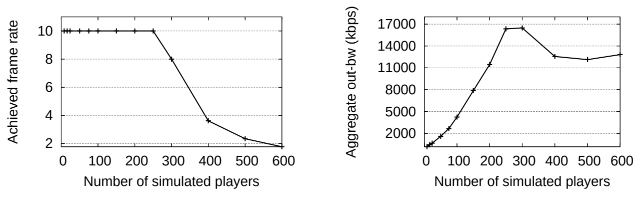
The server implements a discrete event loop in which it invokes the think function for each object in the game and sends out the new view (also called a *frame* in game parlance) of the game world state to each player. In FPS games, this loop is executed 10 to 20 times a second; this frequency (called the *frame-rate*) is generally lower in other genres.

2.2 Client-Server Scaling Properties

The most significant drawback of this client-server game architecture is its reliance on a single server. The server can become a computation and communication bottleneck. To quantify these bottlenecks, we describe the general scaling properties of games and present the scaling measurements from a typical client-server FPS game, Quake II.

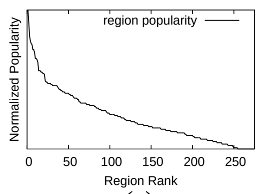
Scalability Analysis: The three game parameters that most impact network performance are: the number of objects in play ($\mathit{NumObjs}$), the average size of those objects ($\mathit{ObjSize}$), and the game's frame-rate ($\mathit{UpdateFreq}$). In Quake II, if we only consider objects representing players (which tend to dominate the game update traffic), $\mathit{NumObjs}$ ranges from 8 to 64, $\mathit{ObjSize}$ is $\sim$200 bytes, and $\mathit{UpdateFreq}$ is 10 updates per second. A naïve server implementation which simply broadcasts the updates of all objects to all game clients ($\mathit{NumClients}$) would incur an outbound bandwidth cost of $\mathit{NumClients} \times \mathit{NumObjs} \times \mathit{ObjSize} \times \mathit{UpdateFreq}$, or 1-66Mbps in the case of Quake II games between 8 and 64 players.

Two common optimizations used by games are *area-of-interest* filtering and *delta-encoding*. Individual players typically only interact with or see a small portion of the game world at any one time. Servers need only update clients about objects in this area-of-interest, thus, reducing the number of objects transferred from the total set of objects ($\mathit{NumObjs}$) to the number of objects in this area ($\mathit{NumAoiObjs}$). If we only consider players, $\mathit{NumAoiObjs}$ is typically about 4 for most Quake II games. Additionally, the set of objects and their state change little from one update to the next. Therefore, most servers simply encode the difference (delta) between updates, thus, reducing the number of bytes for each object transferred from the object's size ($\mathit{ObjSize}$) to the average delta size ($\mathit{AvgUpdateSize}$),



(a) (b)

Figure 1: Computational and network load scaling behavior at the server end of client-server system.



10sec Path Length Stats	
Mean	14.8
Std Dev	3.6
Median	15
95 %ile	20

(a) (b)

Figure 2: Quake III workload characteristics. (a) Popularity of different regions in a map. (b) Lengths of paths traveled by players every 10 seconds.

which is about 24 bytes in Quake II. Thus, the optimized outbound server bandwidth cost would be $\mathit{NumClients} \times \mathit{NumAoiObjs} \times \mathit{AvgUpdateSize} \times \mathit{UpdateFreq}$, or about 62-492kbps in the case of Quake II for 8-64 players.

Empirical Scaling Behavior: Figure 1 shows the performance of a Quake II server running on a Pentium-III 1GHz machine with 512 RAM. We vary the number of clients on this server from 5 to 600. Each client is simulated using a server-side AI bot. Quake II implements area-of-interest filtering, delta-encoding and did not rate-limit clients. Each game was run for 10 minutes at 10 frames per second.

As the computational load on a server increases, the server may require more than 1 frame-time of computation to service all clients. Hence, it may not be able to sustain the target frame rate. Figure 1(a) shows the mean number of frames per second *actually* computed by the server, while Figure 1(b) shows the bandwidth consumed at the server for sending updates to clients. We note several points: first, as the number of players increases, area-of-interest filtering computation becomes a bottleneck[2] and the frame rate drops. Second, Figure 1(b) shows that, as the number of players increases, the bandwidth-demand at the server increases more than linearly, since as the number of players increases, players interaction increases (more missiles are shot, for example). Thus, *NumAoiObjs* increases along with *NumClients* resulting in almost quadratic increase in bandwidth. Finally, we note that for large number of players computational load becomes the bottleneck. The huge reduction in frame-rate offsets any increase in bandwidth that might be expected due to an increase in the number of clients. Therefore, we actually see the bandwidth requirements dropping. Although the absolute limits shown can be raised by a factor of 2 or 3 by employing more powerful servers, clearly a centralized server quickly becomes a bottleneck.

2.3 Multiplayer Gaming Workloads

To further understand the requirements of online games, we studied the behavior of human players in real games. We obtained player movement traces from several actual Quake III games played by real players.[3] We observed that players tended to move between popular "waypoint" regions in the map, and that the popularity distribution of way points was Zipf-like, as seen in Figure 2(a), which ranks the regions in a particular map by popularity (i.e., how often players occupy them). This characteristic suggests that load balancing would be an important property of a distributed gaming architecture. Figure 2(b) shows the length of player movement paths in 10 second intervals, given in bucketized map units (the map is 20 units in diameter). Here we see that, despite the popularity of certain regions, players still move around aggressively in short periods of time; the median path length is 15, which is almost the diameter of the map. Hence, a distributed game architecture must be able to adapt to changes in player positions rather quickly.

Our analysis showed that this model fits the game-play across several different maps and Quake III game types (e.g., Death Match and Capture the Flag). We believe this is representative of a FPS games since objectives and game-play do not vary substantially. Although other existing game genres like RPGs may have less aggressive workloads, limitations in existing architectures are at least partially responsible; for example, because of the latency introduced by a lower update rate, most RPGs only allow indirect attacks on enemies (i.e., attacking by right-clicking on a target rather than by aiming and firing like in an FPS). Future games in any genre will likely take advantage of increased interactivity afforded to it, so it is important for distributed game architectures to support the most aggressive games that exist today. Some massively multiplayer games have already attempted to achieve FPS-like interactivity levels (e.g., Planetside [7]).

2.4 Distributed Deployment Opportunities

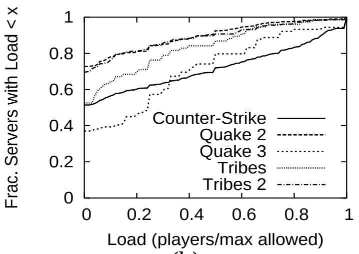
Research designs [32], middle-ware layers [3, 2, 10] and a few commercial games [11, 7] have used server clusters to improve the scaling of server-oriented designs. While this cluster approach is attractive for publishers requiring tight administrative control, we believe that a widely distributed game implementation can address the scaling challenges eliminating possible failure modes. In addition, a dis-

[2]The bot AI code consumed significantly less cycles as compared to the filtering code, and is not a bottleneck in these experiments.

[3]The Quake III server was advertised on the usual master server list and we did not control who actually connected.

Game	*#Servers*
HalfLife / Counter-Strike	10,582
Quake 2	645
Quake 3	112
Tribes	233
Tribes 2	394

(a)



(b)

Figure 3: (a) Observed number of third-party deployed servers for several games, (b) Load on these servers.

tributed design can make use of existing third party federated server deployments that we describe below, which is a significant advantage for small publishers. Such a widely distributed gaming architecture must address unique problems, such as inter-node communication costs and latencies.

There is significant evidence that given appropriate incentives, players are willing to provide resources for multiplayer games. For example, most FPS games servers are run by third parties (such as "clan" organizations formed by players). Figure 3(a) shows the number of active third-party servers we observed for several different games.[4] Older games (e.g., Quake II) typically have much fewer servers than recent ones (e.g., CounterStrike). Figure 3(b) plots the cumulative distribution of load on the different sets of servers, where load is defined as the ratio of the number of active players on the server and the max allowed on the server. Often, more than 50% of the servers have a load of 0. The server count and utilization suggest that there are significant resources that a distributed game design may use.

In this paper, we explore both peer-to-peer designs, which execute the distributed server only on game clients, and federated designs, which can make use of the vast number of publicly available servers for a game.

3 COLYSEUS ARCHITECTURE

In this section, we present an overview of Colyseus, an architecture for distributed multiplayer games. There are two types of game state, immutable and mutable. We assume that immutable state (e.g., map geometry, game code, and graphics) is globally replicated (i.e., every node in the system has a copy) since it is updated very infrequently, if at all. Colyseus manages the collection of mutable objects (e.g., players' avatars, computer controlled characters, doors, items), which we call the *global object store*.

Our architecture is an extension of existing game designs described in Section 2.1. In order to adapt them for a distributed setting, we first must partition mutable state and associated think functions amongst participating nodes. Recall that each game instance runs a synchronous event execution loop. In effect, Colyseus introduces asynchronous *parallel* execution loops, one on each node.

State Partitioning: Each object in the global object store has a *primary* (authoritative) copy that resides on exactly one node. Updates to an object performed on any node in the system are transmitted to the primary owner, which provides a serialization order to updates. If an object had multiple authoritative copies, updates would require running a quorum protocol, incurring intolerable delays since updates occur very frequently. In addition to the primary copy, each node in the system may create a secondary replica (or *replica*, for short). These replicas are created to enable remote nodes to execute code that accesses the object. Replicas are weakly consistent and are synchronized with the primary in an application dependent manner. In practice, the node holding the primary can synchronize replicas the same way viewable objects are synchronized on game clients in client-server architectures. In summary, each node maintains its own *local object store* which is a collection of primaries and replicas of different objects.

Execution Partitioning: As mentioned earlier, existing games execute a discrete event loop that calls the think function of each object in the game once per frame. In our architecture, we retain the same basic design, except for one crucial difference: a node only executes the think functions associated with *primary objects* in its local object store.

Although a think function could access any object in the game world, most think functions do not require access to all of them to execute correctly. Nonetheless, the execution of a think function may require access to objects that a node is not the primary owner of. In order to facilitate the correct execution of this code, a node must create secondary replicas of required objects. Fetching these replicas on-demand could result in a stall in game execution, violating real-time game-play deadlines. Instead, Colyseus has each primary object predict the set of objects that it expects to read or write in the near future, and prefetches the objects in the read and write sets of all primary objects on each node. The prediction of the read and write set for each primary object is specified as a selective filter on object attributes, which we call an object's *area-of-interest*. We believe that most games can succinctly express their areas-of-interest using range predicates over multiple object attributes, which work especially well for describing spatial regions in the game world. For example, a player's interest in all objects in the visible area around its avatar can be expressed as a range query (e.g., $10 < x < 50 \wedge 30 < y < 100$). As a result, Colyseus maintains replicas that are within the union of its primaries' areas-of-interest in each node's local object store.

Object Location: Colyseus uses both traditional randomized DHT and *range-queriable* DHTs as scalable substrates for object location. Range-queries describing area-

[4]Public game servers usually register themselves with a master list; to sample servers, we queried one list for each game and checked that the servers were active.

`class ColyseusObject`	
`GetInterest(Interest* interest)`	Obtain description of object's interests (e.g., visible area bounding box)
`GetLocation(Location* locInfo)`	Obtain concise description of object's location
`IsInterested (ColyseusObject*other)`	Decide whether this object is interested in another
`PackUpdate(Packet* packet, BitMask mask)`	Marshall update of object; bitmask specifies dirty fields for Δ-encoding
`UnpackUpdate(Packet* packet,`	`BitMask mask)` Unmarshall an update for this object

Figure 4: The interface that game objects implement in applications running on Colyseus.

of-interests, which we call *subscriptions*, are sent and stored in the DHT. Other objects periodically publish metadata containing the current values of their *naming* attributes, such as their `x`, `y` and `z` coordinates, in the DHT. We call these messages *publications*. Subscriptions and matching publications are routed to the same *rendezvous* node(s) in the DHT, allowing the *rendezvous* to send all publications to their interested subscribers.

The application of DHTs to a distributed gaming architecture appears straightforward. However, since DHTs have so far been used by applications like storage and bulk data transfer with have relatively "static" workloads, several important challenges arise: What is the most appropriate storage model for quickly changing data items that must be discovered with low latency? Is the load balancing achieved using randomness in DHTs able to cope with the high dynamism and skewed popularity in workloads, or would games benefit substantially from more complex range-queriable DHTs which can better preserve locality and dynamically balance load? Colyseus addresses these and other questions.

Application Interface

From our experience modifying Quake II to use Colyseus (described in Section 7) and our examinations of the source code of several other games, we believe that this model is sufficient for implementing most important game operations. Figure 4 shows the core of the object interface for game objects managed by Colyseus. There are only two major additions to the centralized game programming model, neither of which is likely to be a burden on developers. First, each object uses `GetLocation()` to publish a small number of naming attributes. Second, each object specifies its area-of-interest in `GetInterest()` using range queries on naming attributes (i.e., a declarative variant of how area-of-interest is currently computed).

Colyseus allows nodes to join the system in a fully self-organizing fashion, so there is no centralized coordination or dedicated infrastructure required. Each object is identified by a globally unique identifier or `GUID` (e.g., a large pseudo-random number), and each node is uniquely identified by a routable end-point identifier or `EID` (e.g., an IP address). Applications can use `GUID`s as handles to refer to remote objects.

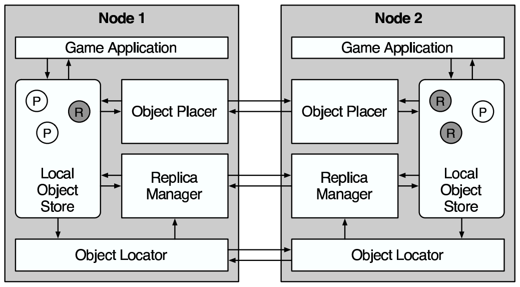


Figure 5: The components of Colyseus. Circled R's represent secondary replicas, circled P's represent primary objects.

This architecture does not address some game components, such as content distribution (e.g., game patch distribution) and persistent storage (e.g., storing persistent player accounts). The problem of distributing these components is orthogonal to distributing game-play and is readily addressed by other research initiatives [16, 18].

Architecture Components

Colyseus implements the preceding design with three components: A *replica manager* (Section 5), which maintains and synchronizes replicas with primaries, an *object locator* (Section 6), which implements a range-queriable lookup system for locating objects within area-of-interests, and an *object placer*, which decides where to place objects and migrate them during game-play. Although all three components are implemented in Colyseus, we focus on the first two components in this paper. We assume an object is placed on the closest node to the controlling player, which is likely optimal for minimizing interactive latency. Figure 5 shows the interaction between each component, the game application, and components on other nodes.

4 EVALUATING DESIGN DECISIONS

In order to evaluate design decisions in Colyseus, we developed our own distributed game based on the characteristics observed in Section 2.3. This section describes this initial workload and the experimental setup of micro-benchmarks we use in the subsequent sections to illustrate important aspects of Colyseus's design. In Section 7, we apply Colyseus in a distributed version of Quake II, demonstrating that our observations apply to a real world game.

4.1 Model Workload

We derived a *model workload* from our observations in Quake III games (see Section 2.3), which we implemented

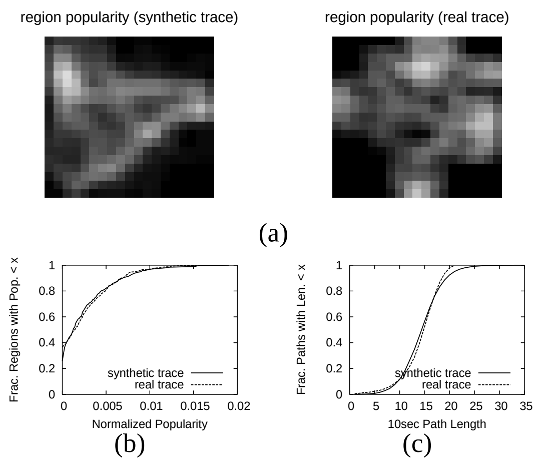


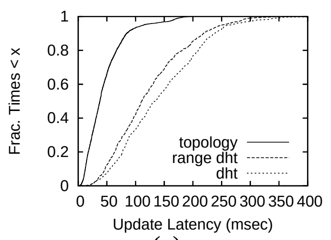
Figure 6: Comparison of our synthetic game trace with a real Quake III trace measured with human players.

as a real game played by bots that runs on top of Colyseus. We randomly generated artificial obstacles on our synthetic maps and used a obstacle-sensitive mobility model from ad-hoc networking research based on Voronoi diagrams [26], so simulated players behave as if obstacles existed. Mobility characteristics like the probability of entering fights and staying at or leaving waypoints were based on trace values. In addition, area of interests are based on median interest sizes observed in Quake II and Quake III maps. Game mechanics such as object velocity, map size, and fight logic were based directly on values from Quake II and Quake III.

Figure 6 compares a trace based on our model workload with a real Quake III trace on a similar map. Part (a) shows the relative popularity of different regions in each map (lighter regions are more popular), where popularity is defined as how often players enter a given region. Although the maps are clearly different, we see that they share similar characteristics, such as several highly popular areas and less popular paths that connect them. Part (b) and (c) compare the distribution of region popularities and lengths of paths (in the number of regions) taken by players/bots during 10 second intervals, respectively. The distributions match up quite closely, with the only non-trivial differences being that our synthetic trace does not have as many 0-popularity regions (due to lack of real obstacles) and have a small number of longer paths. These small differences will tend to reduce the locality of objects, making it slightly more difficult for our designs to cope with the workload.

4.2 Experimental Setup

We emulate the network environment by running several virtual servers on 5-50 physical machines on the Emulab network testbed [35]. The environment does not constrain link capacity, but emulates end-to-end latencies (by approximately delaying packets) using measured Internet pairwise



(a)

% Direct Updates	
Object Type	*%*
Player	98.97
Missile	64.80
All	91.69

(b)

Figure 7: (a) Comparison of update latencies when sent directly through the topology and through a DHT. (b) Percentage of object updates that can bypass object location in a 768-player game.

latencies sampled from the King P2P dataset [6]. Median round trip latencies for samples we used from the dataset are between 80ms-90ms. Due to limited resources and to avoid kernel scheduling artifacts, when running several virtual servers on the same physical machine, we artificially dilate time in our experiments (so that 1 experimental minute lasts 3 actual minutes, using a dilation factor of 3, for example) by dilating all inter-node latencies, timers, and timeouts accordingly.[5] Hence, our latency results do not include computational delays. Our configurations only emulated at most 8 players per server, so computational delay would be negligible even in a real game (e.g., see Quake II in Figure 1(a)). Each game run lasted 8 minutes, which is about half the time of a typical FPS game round. In the following sections, we describe the details of the replica manager and object locator, using the above setup to quantify important points.

5 REPLICA MANAGEMENT

The replica management component manages replica synchronization, responds to requests to replicate primaries on other nodes, and deletes replicas that are no longer needed.

Decoupling Location and Synchronization: An important aspect of Colyseus's replica manager is the decoupling of object discovery and replica synchronization. Once a node discovers a replica it is interested in, it synchronizes it directly with the primary from that point on. A simpler strategy would have been to route updates directly to interested parties though the DHT (such as in [29]). This approach adds at least one extra hop for each update (updates must travel to the rendezvous and then to the interested party), which adds 50-200ms of additional delay 50% of the time (in a traditional DHT with random keys). To quantify the impact of decoupling, Figure 6(a) compares the one-way direct latencies between 96 nodes in a real world end-host topology [6] (topology) and the delivery latency of publications and subscriptions in a 768-player game under our model workload

[5]We use UDP as our underlying transport protocol (doing necessary retransmission at the application layer), so the impact time dilation would have on TCP does not affect our results.

Proactive Replication Mean % Missing Missiles			
Nodes	*Players*	*On*	*Off*
28	224	27.5	72.9
50	400	23.9	64.5
96	768	27.2	72.9

Table 1: Impact of proactive replication on missile object inconsistency.

using both a range-queriable DHT (range dht) and a traditional DHT (dht). Although routing through either substrate achieves much better than $\log n$ hops thanks to the effectiveness of route caching with a highly localized workload, the delays are significantly worse than sending updates directly point-to-point, especially considering the target latency of 50-100ms for many games [12].

The only time when a node must incur the DHT latency is when it must first discovery an object which it does not have a replica of. This occurs when the primary just enters the area-of-interest of a remote object. Figure 7(b) quantifies how often this happens in the same 768 player game, assuming that each player is on a different node (the worst case). For each object type, the table shows the percentage of updates to objects that were previously in a primary's area-of-interest (and hence would already be discovered and not have to incur the lookup latency), as opposed to objects that just entered. For player objects almost 99% of all updates can be sent to replicas directly. For missiles, the percentage is lower since they are created dynamically and exist only for a few seconds, but over half the time missile replicas can still be synchronized directly also. Moreover, enabling interest prediction would further increase the number of updates that do not need to be preceded by a DHT lookup, since nodes essentially discover objects before they actually need them.

Finally, routing updates through the DHT incurs additional communication overhead and complicates traditional optimizations for update delivery in online games. For example, the state required for delta-encoding is simple to maintain when a direct connection is established between the two nodes exchanging deltas, but would require additional connection state at the rendezvous if delta-encoded updates were routed through a DHT. Since the rendezvous changes as an object moves, migration of this state would also be required.

Proactive Replication: To locate short-lived objects like missiles faster, Colyseus leverages the observation that most short-lived objects originate at locations close to their creator, so nodes interested in the creator will always be interested in the new objects. For example, a missile originates in the same location as the player that shot it. Colyseus allows an object to *attach* itself to others (using application annotations). Any node interested in the latter will automatically replicate the former, circumventing the discovery phase altogether.

Table 1 shows the impact of proactive replication on the fraction of missiles missing (i.e., missiles which were in a primary's object store but not yet replicated) from each nodes' local object store (averaged across all time instances) in our model workload . We see that in practice, this simple addition improves consistency of missiles significantly. For example, in a 50 node/400 player game, enabling proactive replication reduces the average fraction of missiles missing from 64% to 24%. If we examined the object stores' 100ms after the creation of a missile, only 3.4% are missing on average (compared to 28% without proactive replication). The remainder of the missing missiles are more likely to be at the periphery of objects' area-of-interests and are more likely to tolerate the extra time for discovery.

Replica Consistency: Although there can be transient inconsistency between primary and replica state, all replicas eventually converge to the same state due to single-copy serialization. Moreover, since updates are processed each frame, convergence is usually very fast. Bounded inconsistency is usually tolerable in games since there is a fundamental limit to human perception in short time-scales and game clients can extrapolate or interpolate object changes to present players with a smooth view of the game [12]. Figure 7(a), shows that we expect over 90% of nodes will incur update delays of $<$100ms more than 90% of the time. Even in current client-server architectures, the client-side state is always slightly out-of-date with respect to the authoritative server state.

Most game applications can probably resolve frequently occurring conflicts simply and inexpensively. For example, in our distributed Quake II implementation, the only frequent conflict that affected game-play was a failure to detect collisions between solid object on different nodes. We implemented a simple "move-backward" conflict resolution strategy when two objects were "stuck together" which resulted in game-play virtually indistinguishable from that had we actually detected the collision.

Moreover, we note that our design is compatible with a variety of existing gaming and continuous consistency models [20, 36], and known interpolation and extrapolation techniques [30] can be used to predict replica state. Lastly, the replica manager uses techniques similar to the Emerald System [27] to handle distributed object pointers, which we found were rarely required in games like Quake II.

6 LOCATING DISTRIBUTED OBJECTS

To locate objects, Colyseus implements a distributed location service on a DHT. Unlike other location services built on DHTs [15, 33], the object locator in Colyseus must be able to locate objects using range queries rather than exact matches and must handle a continuous and changing query stream from each participant in the system. Moreover, data items (i.e., object location information) change frequently and answers to queries must be delivered quickly to avoid degrading the consistency of views on different nodes in the system. In this section we describe three aspects of the object locator that enables it to meet these challenges. In addition, we describe how Colyseus can leverage *range-queriable* DHTs in its object locator design.

6.1 Location Overview

DHTs [34, 31] enable scalable metadata storage and location on a large number of nodes, usually providing a logarithmic bound on the number of hops lookups must traverse. With a traditional DHT, the object locator bucketizes the map into a discrete number of regions and then stores each publication in the DHT under its (random) region key. Similarly, subscriptions are broken up into DHT lookups for each region overlayed by the range query. When each DHT lookup reaches the rendezvous node storing metadata for that region, it returns the publications which match the original query back to the original node.

Range-queriable DHTs [13, 28] may be an even better fit to a distributed game architecture. Unlike traditional DHTs which store publications under discrete random keys to achieve load balance, a range-queriable DHT organizes nodes in a circular overlay where adjacent nodes are responsible for a *contiguous range* of keys. A range query is typically routed by delivering it to the node responsible for leftmost (or rightmost) value in the range. This node then spreads out the query to other nodes in the range, by constructing an ad-hoc distribution tree. For example, using a range-queriable DHT, the object placer could use the x dimension attribute directly as the key. Since key values are stored continuously on the overlay (instead of randomly), range queries can be expressed directly, instead of having to be broken up into multiple DHT lookups. Moreover, object location metadata and queries are likely to exhibit spatial locality, which maps directly onto the overlay, allowing the object locator to circumvent routing paths and deliver messages directly to the rendezvous by caching recent routes. Finally, since nodes balance load dynamically in a range-queriable DHT to match the publication and subscription distribution, they may be able better handle the Zipf-like region popularity distribution observed in Section 2.

Colyseus implements both these object location mechanisms, and we evaluate the trade-offs of each in Section 6.3.

6.2 Reducing Discovery Latency

Regardless of the underlying DHT substrate, the object locator in Colyseus provides two important primitives to reduce the impact of object discovery latency and overhead.

Interest Prediction and Aggregation: Spatial and temporal locality in object movement enables prediction of subscriptions (e.g., if an object can estimate where it will be in the near future, it can simply subscribe to that entire region as

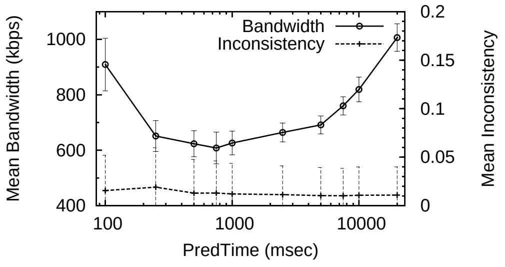


Figure 8: The impact of varying PredTime on total mean node bandwidth and local object store inconsistency.

well). Colyseus expands a the bounding volume subscribed to by an object using the following formula:

$$\text{Vol.min} \mathrel{-}= \text{PredTime} \times \text{PredMoveUpLeft} + \text{PubTime}$$
$$\text{Vol.max} \mathrel{+}= \text{PredTime} \times \text{PredMoveDownRight} + \text{PubTime}$$

This formula predicts the amount of movement an object will make in each direction per game time unit and multiplies it by the desired prediction time (PredTime). We use a simple moving average of an object's velocity to estimate PredMoveUpLeft and PredMoveDownRight, and special cases are made if more is known about an object's physics (e.g., missiles always move in a straight line). A small factor (PubTime) is added to account for the discovery and delivery time of publications for objects entering the object's subscription volume. Subscription prediction amounts to *speculative prefetching* of object location attributes.

Figure 8(a) shows the impact of tuning subscription prediction (by varying PredTime) in a 50 node/400 player game using our model workload (see Section 4 for experimental setup details). The top line plots the total mean bandwidth required by each node, while the bottom line shows the mean local object store inconsistency, which we define in this experiment to be the average fraction of missing player replicas in each node's object store across all time instances (an object is missing if it enters a primary's area-of-interest, but has not been discovered yet). Error bars indicate one standard deviation.

The variation in bandwidth cost as we increase PredTime demonstrates the effects of speculation. When speculation time is too short (e.g., we only predict 100 ms or 1 frame into the future), each object must update subscriptions in the system more frequently, incurring a high overhead. If speculation time is too long, although objects can leave their subscriptions in the system for longer periods of time without updates, they receive a large number of false matches (publications which are in the speculated area-of-interest but not in the actual area-of-interest), also incurring overhead.[6]

[6]Note that extraneous delivery of matched publications need not result in unnecessary replication, since upon reception of a pre-fetched publication, a node can cache (for the length of the TTL) and periodically check whether it *actually* desires the publishing object by comparing the publication to its up-to-date unpredicted

In this particular configuration, the "sweet-spot" is setting PredTime around 1 second. Although this optimal point will vary depending on game characteristics (e.g., density, update size, etc.), notice that we are able to maintain the same level of inconsistency regardless of the PredTime value (in fact, it decreases slightly as we increase PredTime, since some "extra" matches are actually desired). Hence, we can automatically optimize PredTime without affecting the level of inconsistency observed by the game.

To further reduce subscription overhead, Colyseus enables aggregation of overlapping subscriptions using a local *subscription cache*, which recalls subscriptions whose TTLs have not yet expired (and, thus, are still registered in Mercury), and an optional *aggregation filter*, which takes multiple subscriptions and merges them if they contain sufficient overlap. This filter uses efficient multi-dimensional box grouping techniques originally used in spatial databases [23].

Soft State Storage: In most object discovery and publish-subscribe systems implemented on DHTs, only subscriptions are registered and maintained in the DHT while publications are not stored at the rendezvous. The object locator stores both publications and subscriptions as soft state at the rendezvous, which expire them after a TTL carried by each item. When a subscription arrives, it matches with all currently stored publications, in addition to publications that arrive after it.

This design achieves two goals: First, if only subscriptions were stored, subscribers would have to wait until the *next* publication of an interesting object before it would be matched at the rendezvous. By storing publications, a subscription can immediately be matched to *recent* publications. This suffices for informing the node about relevant objects due to spatial locality of object updates. Second, different types of objects change their naming attributes at different frequencies (e.g., items only change locations if picked up by a player), so it would be wasteful to publish them all at the same rate. Moreover, even objects with frequently changing naming attributes can publish at lower rates (with longer TTLs) by having subscription prediction take into account the amount possible staleness (e.g., we can add PubTTL $\times$ Velocity to the PubTime factor above, accounting for how far an object could have moved between publication intervals).

6.3 Comparison of Routing Substrates

In this section, we evaluate how the performance of Colyseus is affected by the choice of the routing substrate, viz. a simple DHT versus a range-queriable DHT. We focus on two types of deployment scenarios: 1) a purely peer-to-peer deployment where each server is co-located with a client, and 2) a federated deployment where the game is deployed on a number of different servers, each hosting multiple clients. We simulate 8 bots per server in the federated scenario. The metrics evaluated include scaling of per-node outgoing bandwidth, latency of object discovery and the impact these have on perceived interactive game-play "lag".

Experimental Setup: Our implementation of Colyseus uses the Mercury protocol described in [13] with the extensions described earlier in this section. Mercury is used both as a simple DHT and as a range-queriable DHT, handling publications and subscriptions as described in Section 6.1. When used as a DHT, Mercury breaks up each map into a number of regions equal to the number of players in a map. When used as a range-queriable DHT, the x dimension is used as the key attribute. In both cases, a cache of size $2\log(n)$ entries is used for caching recently used routes.

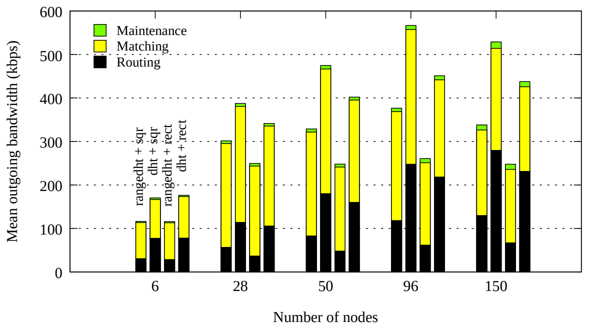
The model workload uses two types of maps: square and rectangular. The height of all rectangular maps is fixed at the standard 16 player map diameter in Quake III, while the width depends on the number of players. We keep the mean player density equal in both kinds of maps. Rectangular maps simulate a linearization (e.g., using Hilbert space-filling curves) of a multi-dimensional map, which may be possible in some games without losing much locality.

6.3.1 *Communication Costs*

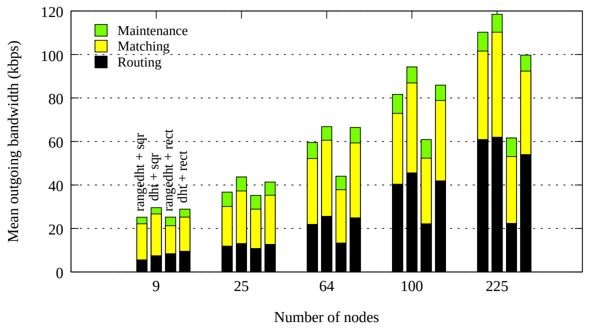
Figure 9 compares the average per-node outgoing bandwidth requirements for the object discovery phase. The bandwidth value reported by each node is the mean taken over a 5-minute period in the middle of the experiment. Bandwidth is divided into two components: sending and routing publications and subscriptions in Mercury (routing) and delivering matched publications and subscriptions (matching). DHT maintenance traffic is also shown (maintenance), but is always low. In general, we find that a rangedht consumes less bandwidth per node as compared to a dht, however the difference varies noticeably in the different map scenarios.

Focusing on the square maps scenario (the first two bars in each set), we see that a rangedht performs much better in the federated scenario but the performance gap almost vanishes in the P2P scenario. For the dht, predicted subscriptions (which can span multiple DHT regions) are sent independently and each incurs a $\log(n)$ hop overhead. On the other hand, for a rangedht, the range subscription incurs the $\log(n)$ hop overhead only for the first hop to reach the leftmost end of the range. After that, it is fanned out to the contiguous set of nodes spanning the range. The latter is more efficient if the range subscription does not span a large number of nodes. This happens more frequently in the federated scenario than in the P2P scenario since each node is responsible for a larger portion of the key-space in the for-

subscriptions locally. Hence overhead is extra received publications.



(a) Federated-servers scenario



(b) Peer-to-peer scenario

Figure 9: Scaling of per-node bandwidth using a dht **and** rangedht**, in square and rectangular maps. Adjacent bars in each group represent** rangedht + sqr**,** dht + sqr**,** rangedht + rect **and** dht + rect **scenarios, respectively. Note the different scales in the two graphs.**

mer.

Performance of a dht is similar for both square and rectangular maps, especially in the P2P scenario. The small reduction is due to the larger boundary of rectangular maps, where subscriptions get clipped. However, a rangedht performs noticeably better in both scenarios with rectangular maps. This is because the total span of the key-space is larger relative to the width of subscriptions, so each subscription covers a fewer number of node ranges.

Scaling Behavior: In general, for a rangedht substrate, as more nodes are added into the system, the number of nodes contacted for each subscription stays constant if using a rectangular map, but grows proportional to $\sqrt{n}$ if using a square map. For a dht substrate, this number stays constant irrespective of the map. However, the lack of contiguity or locality in the generated subscriptions can result in higher routing overhead. In addition to these effects, if player interaction grows as the number of players in the game increase (as in our model), the overall routing traffic will grow (as Figure 9 shows) because more publications are injected into the system. Hence, we observe that both dht and rangedht routing bandwidth scale poorly using square maps, but rangedht scales well with a linearized rectangular map.

Metric		dht	loadbal
Per-node total bwidth (normalized by mean)	std-dev	0.30	0.15
	max	1.93	1.45
Per-node matching bwidth (normalized by mean)	std-dev	1.01	0.57
	max	4.41	2.57
Avg. % missing replicas		8%±6%	10%±9%

Table 2: Effectiveness of a load-balanced rangedht**. 96-node expt. with rectangular maps, federated servers scenario. The percentage of missing replicas shows the mean and standard deviation.**

Load Balancing: Until now, we have only reported the mean per-node bandwidth across all nodes. However, since popularity of the regions in the model workload is Zipfian, nodes in the routing ring responsible for such regions can get considerably more traffic than others. We now focus on the effectiveness of the leave-join load-balancing mechanisms built inside a rangedht like Mercury, which dynamically move lightly loaded nodes to heavily loaded regions the DHT. In our implementation, we use the number of publications and subscriptions routed per second, averaged over a 30-second moving window, as a measure of the load.

Table 2 compares the bandwidth and view inconsistency (Section 6.3.2) for a 96-node experiment in the federated scenario. We find that with load-balancing turned on, Mercury reduces the maximum total mean-bandwidth by about 25% (relative to the mean) and the maximum matching mean-bandwidth by about 42%, compared to a dht. While partitioning a busy range may not necessarily result in decreasing routing load (since subscriptions spanning the range will have to visit both nodes), it is effective at partitioning the matching load which is a significant component of the total bandwidth costs (see Figure 9). Also, the average fraction of missing replicas (Section 6.3.2) is not substantially higher, suggesting that players do not lose many updates due to leave-join movements in the routing ring.

6.3.2 *Latency and Inconsistency*

Recall that Colyseus needs to discover and fetch objects in a player's area-of-interest as the player is moving rapidly through the game world. If latencies of more than a few frames are incurred, a player's view may be rendered incorrectly. In this section, we evaluate the impact of the routing substrate on game-state consistency. To measure the inconsistency penalty incurred by distributing a game, we first evaluate how long it takes for a node to discover and replicate an object that it is interested in. This provides an estimate of the worst case delay that a view might have to endure. We then examine the impact that the latency has on the consistency of local object stores on different nodes.

Discovery Latency: Figure 10 shows the median time elapsed between submitting a subscription and generation of a matching publication for different routing policies and map types. This latency is broken down into two parts: (1)

routing the subscription to the *first* (left-most) rendezvous (routing), and (2) delay incurred at the rendezvous before a matching publication arrives (matching). To completely construct a replica, an additional average delay of $1.5 * \texttt{RTT}$ must be added: $0.5 * \texttt{RTT}$ for delivering the publication, and $1.0 * \texttt{RTT}$ for fetching the replica. However, this latency is independent of the location substrate. In Figure 10, we show results for the P2P scenario; results for the federated scenario are similar.

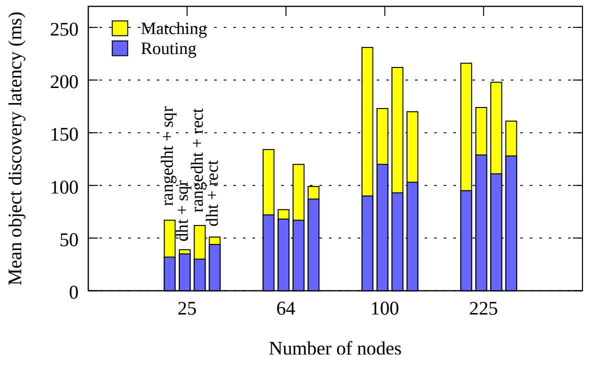


Figure 10: The mean time required to discover replica of an object once a subscription is generated in the P2P scenario.

We find that the routing delay for subscriptions scales similarly for both routing policies, as expected. Both policies are able to exploit caching quite well with median hop length $<= 3$ for all cases. However, we find that the matching latency is higher for the rangedht case. This is because the matching component incorporates the latency incurred when spreading the range subscription *after* reaching the left-most rendezvous point. Notice that, for square maps, the number of nodes spanning a subscription increases, and so does the matching latency for the rangedht. A dht thus incurs bandwidth overhead by sending multiple disjoint subscriptions, but obtains an small overall latency advantage.

Although interactive latencies exceeding 50-100ms may be noticeable by players [12], discovery latencies are only incurred upon first discovering an interesting object. For example, when a player enters a new room or another player comes into the periphery of a player's visible area. Once a replica is discovered and created, it will be kept up to date through direct communication with the primary. Hence replica staleness will be tied to the latency distribution of the topology, which we saw was low in Section 5. Note that our Mercury implementation did not incorporate proximity routing techniques proposed recently for DHTs [22, 17], which may further reduce the latency of the routing component in both cases.

Replica View Inconsistency: To examine the impact that object discovery latency has on object store consistency, we examine how often replicas within the area-of-interest of a node are not present in the node's object store. We compute the fraction of replicas missing at a given time instance as the

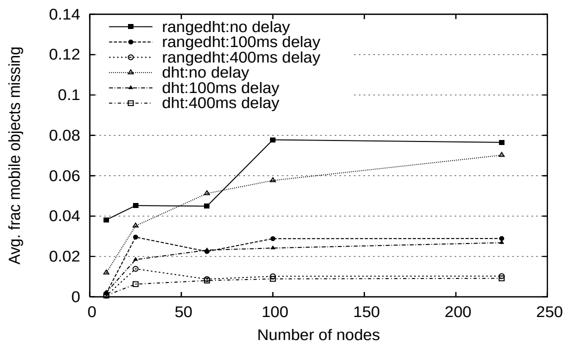


Figure 11: The fraction of replicas missing averaged across all time instances as we scale the number of servers in the peer-to-peer scenario (square maps).

ratio of the number of missing replicas and the total number of replicas required (counting only time instances where the node requires at least 1 replica). Figure 11 shows the fraction of replicas missing for a dht and rangedht, if we allow 0ms, 100ms and 400ms to elapse after a node required them. We show results only for the P2P scenario, but results for the federated case are similar.

We see that inconsistency in game state is approximately the same irrespective of the choice of the routing substrate. The rangedht has slightly higher inconsistency due to the higher object discovery latency. However, this difference vanishes if we allow for a small delay of 100ms. For both policies, the inconsistency is fairly low. For example, with 64 nodes, about 4% of the objects required are missing at any given time. This improves to about 2% missing if we allow for a 100ms delay (1 frame), and it improves to only about 1% missing if we allow for a 400ms delay (4 frames). Note that fractional values exaggerate the absolute number of replicas missing, since only 3.38 replicas were required by each node (on average) in this experiment.

Summary

To summarize, we find that a rangedht incurs lower bandwidth overhead compared to a dht by utilizing contiguity in data placement. Although a rangedht incurs higher object discovery latency compared to a dht, at time scales of 50-100ms, the resultant inconsistency in game-state is indistinguishable. Finally, given the promising load balancing results for a rangedht, we believe that a single-attribute rangedht (using linearization of a 2D or 3D map) is most suitable as a routing substrate for Colyseus.

7 EVALUATION WITH A REAL GAME

To demonstrate the practicality of our system, we modified Quake II, a popular first person shooter game, to use Colyseus. We describe our experience with this distributed Quake II server in this section.

In our Quake II implementation, we represent an object's area-of-interest with a variable-sized bounding box encompassing the area visible to the object. We automatically delta-encode and serialize Quake II objects using field-wise diffs, so the average object delta size in our implementation is 145 bytes. Quake II's server to client messages are more carefully hand-optimized and average only 22 bytes. Unmodified Quake II clients can connect to our distributed servers and play the game with an interactive lag similar to that obtained with a centralized server. As a result, the system can be run as a peer-to-peer application (with every client running a copy of the distributed server) or as a distributed community of servers.

We used a large, custom map with computer controlled bots for players as the workload for our Quake II evaluation. We also used the Emulab testbed setup described in section 4. However, we did *not* artificially dilate time, so all numbers reported take into account actual execution times. We use Mercury as the object location substrate. Details about our Quake II prototype and additional evaluation results can be found in an associated technical report [14].

7.1 Communication Cost

Figure 12 compares the bandwidth scaling of Colyseus running in a peer-to-peer scenario with the client-server and broadcast architecture alternatives. We simulate the alternatives using the same game-play events as the real execution on Colyseus. In the client-server case, a single server handles all clients. In the broadcast scenario, each client broadcasts its updates to all other clients.

Figure 12(a) shows the scaling properties under our model workload. The workload keeps player density constant by increasing the map size. The thin error bar indicates the 95th percentile of 1 second burst rates across all nodes, while thick error bars indicate 1 standard deviation from the mean. The colyseus and broadcast lines show per-node bandwidth while the colyseys-aggregate line shows the total bandwidth used by all nodes in the system. At very small scales (e.g., 9 players), the overhead introduced by object location is high and Colyseus performs worse than broadcast. As the number of nodes increases, each node in Colyseus generates an order of magnitude less bandwidth than each broadcast node or a centralized server. Moreover, we see that Colyseus's per-node bandwidth costs rise much more slowly with the number of nodes increase than either of the alternatives. Nonetheless, the colyseus-aggregate line shows that we do incur an overall overhead factor of about 5. This is unlikely to be an issue if each node as sufficient resources.

Figure 12(b) shows the same figure when running with the Quake II workload. We observe similar scaling characteristics here, except that the per-node Colyseus bandwidth appears to scale almost quadratically rather than less-than-linearly as in our model workload. This is primarily due to the fact that the Quake II experiments were run on the same-sized map. Thus, the density of the game (and hence number of visible objects per player) increased with the number of players/nodes, which adds a quadratic scaling factor to all four lines. To account for this effect, Figure 12(c) shows how each component of Colyseus's traffic scales (per node) if we fixed the number of players in the map at 400 and increase the number of server nodes handling those players (by dividing them equally among the nodes). Due to inter-node interests between objects, increasing the number of nodes by a factor of k may not reduce per-node bandwidth cost by the same factor. For example, we see a 3-fold decrease in communication cost per node with a 5-fold increase in the number of nodes, so this overhead is less than a factor of 2. We expect similar bandwidth scaling characteristics to hold for our model workload and Quake II if player density were fixed at a reasonable level for the game size. In addition, this result shows that the addition of resources in a federated deployment scenario can effectively reduce per-node costs.[7]

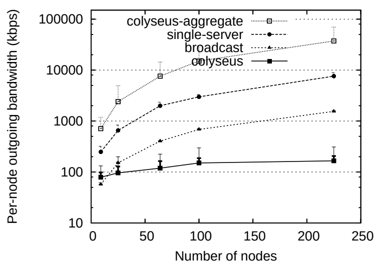
7.2 View Inconsistency

We now examine the view inconsistency (i.e., fraction of desired replicas that are missing from a node's local object store) observed in the Quake II workload, like we did for the model workload in Section 6.3.2.[8] Figure 13(a) shows the fraction of replicas missing as we scale the number of nodes in the peer-to-peer Quake II scenario. We note that these results are very similar to the inconsistency we observed with the model workload and, in the no delay case, achieves around the 9% inconsistency we expected from updates that require a preceding object discovery seen in Figure 7(b). Moreover, nearly one half of the replicas a node is missing at any given time instance arrives within 100ms and less than 1% take longer than 400ms to arrive.

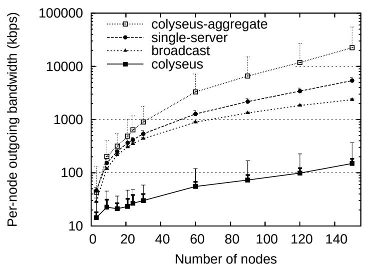
Figure 13(b) shows the cumulative distribution of the number of missing objects for a 40-node (320-players) federated servers scenario. On average, a node requires 23 remote replicas at a given time instance. About 40% of the time, a node is missing no replicas; this improves to about 60% of the time if we wait 100ms (or one frame in Quake II) for a replica to arrive and to over 80% of the time if we wait 400ms for a replica to arrive. We are missing more than 1 replica less than 35% of time; 20% of the time if we wait 100ms and 5% of the time if we wait 400ms. The inconsistency is less for sparser game playouts. For example, in

[7] If we had hand-tuned update delta sizes so they averaged 22 bytes instead of 145 bytes like in these experiments, the client-server and broadcast architectures would perform better. However, note that Figure 12(c) shows that updates also account for over 75% of Colyseus's costs, so Colyseus would get almost the same benefit as well. Moreover, the scaling properties would not change.

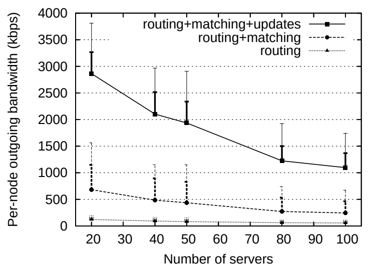
[8] We ignore non-mobile objects like items in this analysis. Including them would increase overall consistency, since they do not move.



(a) Model Workload

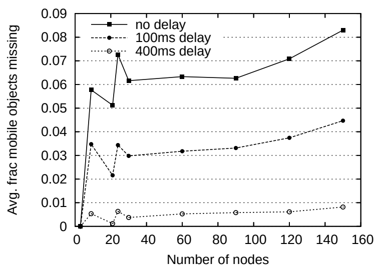


(b) Quake II

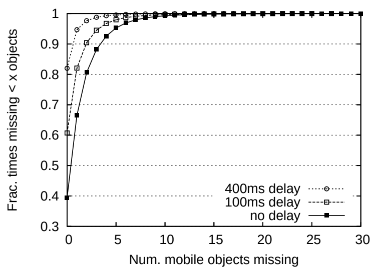


(c) Quake II (Same Density)

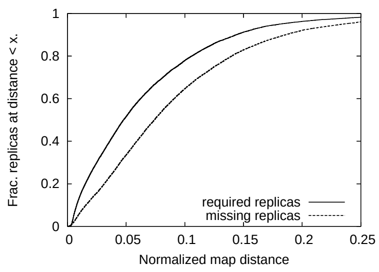
Figure 12: The bandwidth scaling properties of Colyseus using (a) the model workload and (b) the Quake II workload, compared with client-server and broadcast architectures (note the logarithmic scale). Part (c) shows the scaling of Quake II, keeping the player density the same.



(a) Consistency Scaling



(b) 40 Nodes/320 Players



(c) Missing Replica Locations

Figure 13: (a) Mean fraction of replicas missing as we vary the number of servers/players in Quake II. (b) CDF of missing objects in a 40 Nodes/240 Players. (c) CDF showing the distance of missing replicas from a subscriber's origin.

the 150 node peer-to-peer scenario (150 players), no objects were missing 84% of the time and 98% of the time after 400ms. Figure 13(c) compares the distance of replicas to the all objects in the area-of-interest to the distance of those that are missing, across all time instances in the game. Replicas that are missing from a view tend to be closer to the periphery of object subscriptions (and hence, father away from the interested object and less important). Although the difference in the distributions is not substantial, subscription sizes in our Quake II workload are variable, so objects at the periphery of a subscription may still be close to a player if they are in a small room.

8 RELATED WORK

In this section, we briefly comment on the designs adopted by current games, as well as previous research architectures for distributed games.

Some games (e.g., MiMaze [20], Halo [5], and most Real Time Strategy (RTS) games [1]) use a *parallel simulation* architecture, where each player in the game simulates the entire game world. All game objects are replicated everywhere and kept consistent using lock-step synchronization. The obvious disadvantages of this architecture are its broadcasting of updates to every player, resulting in quadratic bandwidth scaling behavior, and its need for synchronization, limiting response time to the speed of the slowest client. These deficiencies are tolerated in RTS games because individual games rarely involve more than 8 players.

Second-Life [11] and Butterfly.net [3] perform interest filtering by partitioning the game world into disjoint regions or cells. Much like Colyseus' DHT-based lookup, SimMUD [29] makes this approach fully distributed by assigning cells to keys in a DHT. Zou, et al. [37] theoretically compare cell-centric approaches with entity-centric approaches, like Colyseus. While our results show that such a design works well, it does have some weaknesses. The granularity of the cells must be chosen to carefully match the typical area-of-interest size. This may be difficult in some games where the area-of-interest size varies widely.

Furthermore, while the above approaches share some commonalities with our design, we believe we are the first to demonstrate the feasibility of implementing a real-world game on a distributed architecture that is not designed for a centralized cluster (like Second-Life and Butterfly.net). Colyseus is also able to support FPS games which have much tighter latency constraints than RPGs (which were targeted by SimMUD, for example.)

Several architectures proposed for Distributed Virtual Reality (VR) environments and distributed simulation (notably, DIVE[19], MASSIVE [21], and High Level Architecture (HLA) [24]) have similar goals as Colyseus but focus on different design aspects. DIVE and MASSIVE focus on sharing audio and video streams between participants while HLA is designed for military simulations. None address the specific needs of modern multiplayer games and, to our knowledge, none have been demonstrated to scale to large numbers of participants. For example, DIVE and HLA originally assumed wide-scale deployment of IP-Multicast.

9 SUMMARY AND FUTURE WORK

In this paper, we have described the design, implementation and evaluation of Colyseus, a distributed architecture for online multiplayer games. Colyseus takes advantage of a game's ability to tolerate inconsistent state in its partitioning of state across system nodes. It also takes advantage of game software's predictable read/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-queriable DHT is a better choice as the routing substrate as compared to a normal DHT, predominantly due to its load-balancing properties. Our adaptation of a commercial game (Quake II) demonstrated the practicality of Colyseus' design.

Nonetheless, we think of Colyseus as on-going work since the current design does not address some important problems in a distributed setting. Our current implementation does not tolerate node departures, but we believe that the presence of a large number of object replicas in our current design can be leveraged to provide continuity after node failures. Furthermore, note that the Mercury location component can withstand moderate node churn rates. Finally, a distributed architecture also raises issues about cheating in a peer-to-peer setting. Addressing cheating will need to be a critical part of any distributed architecture for it to be practical. To address these problems, we believe we can leverage Colyseus' flexibility in object placement. For example, by carefully selecting the owners of primary objects, we may be able to limit the damage malicious players or nodes can inflict on others. We plan to address these challenges in future work.

For more information about the project (software, documentation and announcements), please visit: `http://www.cs.cmu.edu/~ashu/gamearch.html`

REFERENCES

[1] Age of Empires. `http://www.microsoft.com/games/empires/`.

[2] Big World. `http://www.microforte.com/`.

[3] Butterfly.net. `http://www.butterfly.net/`.

[4] Everquest Online. `http://www.everquest.com`.

[5] Halo. `http://www.xbox.com/en-US/halo/`.

[6] King peer-to-peer measurements data set. `http://www.pdos.lcs.mit.edu/p2psim/kingdata`.

[7] PlanetSide. `http://planetside.station.sony.com`.

[8] QuakeII Game Engine v3.20. `ftp://ftp.idsoftware.com/idstuff/quake2`.

[9] Torque Networking Library. `http://www.garagegames.com`.

[10] Zona. `http://www.zona.net/`.

[11] Enabling Player-Created Online Worlds with Grid Computing and Streaming. `http://www.gamasutra.com/resource_guide/20030916/rosedale_01.shtml`, 2003.

[12] BEIGBEDER, T. ET AL. The Effects of Loss and Latency on User Performance in Unreal Tournament 2003. In *NetGames 2004*.

[13] BHARAMBE, A., AGRAWAL, M., AND SESHAN, S. Mercury: Supporting scalable multi-attribute range queries. In *SIGCOMM 2004*.

[14] BHARAMBE, A., PANG, J., AND SESHAN, S. A Distributed Architecture for Interactive Multiplayer Games. Tech. Rep. CMU-CS-05-112, Carnegie Mellon University, Jan. 2005.

[15] CABRERA, L. F., JONES, M. B., AND THEIMER, M. Herald: Achieving a Global Event Notification Service. In *HotOS 2001*.

[16] DABEK F., ET AL. Wide-area cooperative storage with CFS. In *SOSP 2001*.

[17] DABEK, F., LI, J. ET AL. Designing a DHT for low latency and high throughput. In *NSDI 2004*.

[18] DRUSCHEL, P., ROWSTRON, A. Storage mgmt. and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP 2001*.

[19] FRÉCON, E., AND STENIUS, M. DIVE: A scaleable network architecture for distributed virtual environments. *Distributed Systems Engineering Journal 5*, 3 (1998), 91–100.

[20] GAUTIER, L., AND DIOT, C. MiMaze, A Multiuser Game on the Internet. Tech. Rep. RR-3248, INRIA, France, Sept. 1997.

[21] GREENHALGH, C., AND BENFORD, S. Massive: a distributed virtual reality system incorporating spatial trading. In *ICDCS 95*.

[22] GUMMADI, K. P., ET. AL. The Impact of DHT Routing Geometry on Resilience and Proximity. In *SIGCOMM 2003*.

[23] GUTTMAN, A. R-trees: a dynamic index structure for spatial searching. In *SIGMOD 1984*.

[24] IEEE. High Level Architecture. `http://www.dmso.mil/public/transition/hla/index.html`.

[25] IEEE. Standard for Information Technology, Protocols for Distributed Interactive Simulation. Tech. rep., Mar. 1993.

[26] JARDOSH, A. ET AL. Towards Realistic Mobility Models for Mobile Ad hoc Network. In *MOBICOM 2003*.

[27] JUL, E. ET AL. Fine-grained mobility in the emerald system. *ACM Trans. Comput. Syst. 6*, 1 (1988), 109–133.

[28] KARGER, D., AND RUHL, M. Simple efficient load-balancing algorithms for peer-to-peer systems. *IPTPS 2004* (2004).

[29] KNUTSSON, B. ET AL. Peer-to-peer support for massively multiplayer games. In *IEEE INFOCOM 2004*.

[30] PANTEL, L., AND WOLF, L. C. On the suitability of dead reckoning schemes for games. In *NetGames 2002*.

[31] ROWSTRON, A., DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale p2p systems. In *Middleware 2001*.

[32] SHAIKH, A., SAHU, S. ET AL. Implementation of a Service Platform for Online Games. In *NetGames 2004*.

[33] STOICA, I., ADKINS, D. ET AL. Internet Indirection Infrastructure. In *SIGCOMM 2003*.

[34] STOICA, I., MORRIS, R. ET AL. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM 2001*.

[35] WHITE, B., LEPREAU, J. ET AL. An integrated experimental environment for distributed systems and networks. In *OSDI 2002*.

[36] YU, H., VAHDAT, A. Design and Evaluation of a Conit-based Continuous Consistency Model for Replicated Services. *ACM TOCS* (2002).

[37] ZOU, L., AMMAR, M. ET AL. An evaluation of grouping techniques for state dissemination in networked multi-user games. In *MASCOTS 2001*.