Colyseus: A Distributed Architecture for Online Multiplayer Games

Ashwin Bharambe
Carnegie Mellon University
ashu+@cs.cmu.edu

Jeffrey Pang
Carnegie Mellon University
jeffpang+@cs.cmu.edu

Srinivasan Seshan
Carnegie Mellon University
srini@cmu.edu

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. In addition, it 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 Quake II, a popular first person shooter game, to use it. Our measurements of Quake II and our own Colyseus-based game with hundreds of players shows that Colyseus effectively distributes game traffic across the participating nodes, allowing Colyseus to support low-latency game-play for an order of magnitude more players than existing single server designs, with similar per-node bandwidth costs.

1 Introduction

Networked games are rapidly evolving from small 4-8 person, one-time play games to large-scale games involving thousands of participants and persistent game worlds. Almost all networked games, however, are centralized — players send control messages to a central server and the server sends relevant state updates to all active players. This approach 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 in 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 nor 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., game objects) and execution (e.g., the logic to simulate player and game AI actions) among the participating nodes. Distributing a

networked game is even more difficult due to 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, we can take advantage of two fundamental properties of games to address these challenges. First, games tolerate weak consistency in the application state. For example, 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 and writes to the shared state highly predictable. For example, most reads and writes by a player occur upon objects that are physically close to that player in the game world. 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 updates to an object are serialized through exactly one primary copy in the system. This approach mirrors the consistency model of existing client-server architectures on a per object basis. Although replicas are only kept weakly consistent with the primary copy, they enable the low-latency read access needed to keep game execution timely. The challenge is for each node to determine the set of replicas it needs in advance of executing any game logic. Colyseus provides a rich query interface over the system-wide collection of objects to identify and fetch required objects. We have implemented this query interface on both a randomized distributed hash table (DHT) [28] and a dynamically load balanced, range-based DHT [3]. However, lookups in DHTs can be too slow for finding required replicas in games. To hide this lookup latency, Colyseus uses locality and predictability in data access patterns to speculatively pre-fetch objects. This mechanism is only used to *discover* relevant objects; updates are propagated from primary copies to

replicas directly. 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 modified Quake II [22], a popular server-based First Person Shooter (FPS) game, to run on our implementation of Colyseus, and have also used measurements of Quake III [23] game-play to develop our own Colyseus-based game with players that mimic the Quake III workload. These concrete case studies illustrate the practicality of using our architecture to distribute existing game implementations. Our measurements on an Emulab testbed with hundreds of players show that Colyseus is effective at distributing game traffic and workload across the participating nodes, while providing servers and players with low-latency and consistent views 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.

2.1 Contemporary Game Design

To determine the requirements of multiplayer games, we studied the source code of several popular and publicly released engines for *virtual reality* games, including Quake II [22], Quake III [23], and the Torque Networking Library [30]. In these games, each *player* (game client) controls an *avatar* (player's representative in the game) in a large *game world*, though a player only interacts with a small portion of the world at any given time. This description applies to many popular genres, including FPSs (such as *Quake* and *Counter Strike*), role playing games (RPGs) (such as *Everquest* and *World of Warcraft*), and others. There are certainly some game genres that do not fit this description, such as Real Time Strategy (RTS) or puzzle games, but they are outside the scope of our study.

Almost all commercial virtual reality games are based on a client-server architecture where a single server maintains the state of the game world or disjoint portion of the game world. 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 its 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.

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

2.2 Client-Server Scaling Properties

The single server hosting a game can become a computation and communication bottleneck. To quantify these bottlenecks, we describe the general scaling properties of games and present measurements from Quake II, a typical FPS game.

Scalability Analysis: A game server's outbound bandwidth requirement (which is substantially higher than its inbound traffic [10]) is mainly determined by three game parameters: number of objects in play n, the average size of those objects s, and the game's frame-rate f. For example, in Quake II, if we only consider objects representing players (which tend to dominate game update traffic), n ranges from 8 to 64, s is about 200 bytes, and f is 10 updates per second. A naïve server implementation which simply broadcasts the updates of all objects to all c game clients would incur an outbound bandwidth cost of $c \cdot n \cdot s \cdot f$, or 1-66Mbps in games with 8 to 64 players.

Two common optimizations are employed to reduce this cost: *area-of-interest* filtering and *delta-encoding*. Since individual players only interact with a small portion of the game world at any given time, only updates about *relevant* objects are sent to the clients. Additionally, object state changes little from one update to the next. Therefore, most servers send the difference (i.e., delta) between updates. These optimizations reduce n and s respectively. In the case of 8-64 player Quake II games, the server bandwidth requirement reduces to about 62-492 kbps.

Empirical Scaling Behavior: Figure 1 shows the performance of a Quake II server running on a Pentium-III 1GHz machine with 512 RAM with different numbers of players. Each player is simulated using a server-side AI bot (though the server sends packets to them as if they were real clients). The server implements area-of-interest filtering, delta-encoding and does not rate-limit

(a)   (b)
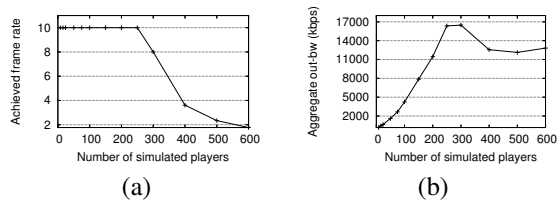
Figure 1: Computational and network load scaling behavior of a Quake II server.

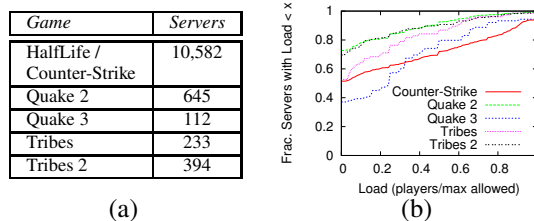| 10sec Path Length Stats	
Mean	14.8
Std Dev	3.6
Median	15
95 %ile	20

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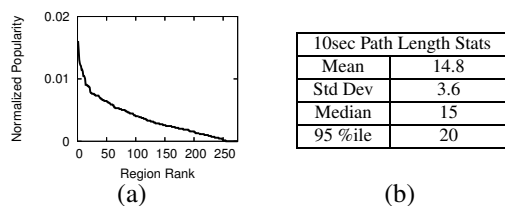
Figure 2: Quake III workload characteristics. (a) Popularity of different regions in a map. (b) Lengths of paths traveled by players every 10 seconds.

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 and the frame-rate drops. (Detailed measurements show that the computational bottleneck is indeed the filtering code and not our AI bot code.) 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, player interaction increases (for example, more missiles are fired.) Thus, n increases along with c resulting in a super-linear increase in bandwidth. Finally, we note that when the number of players exceeds 250, computational load becomes the bottleneck. The reduction in frame-rate offsets the increase in per-frame bandwidth (due to the increase in the number of clients), so we actually see the bandwidth requirement decrease. Although the absolute limits can be raised by employing a more powerful server, this illustrates that it is difficult for any centralized server to handle thousands of players.

2.3 A Multiplayer Gaming Workload

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 ac-

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.

tual Quake III games played by actual players on an Internet server. We observed that players tended to move between popular "waypoint" regions in the map and the popularity distribution of waypoints was Zipf-like. Figure 2(a) 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.) 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 quickly.

Our analysis showed that this model fits the game-play across several different maps and game types (e.g., Death Match and Capture the Flag), and we believe that it is representative of other FPS games since objectives and game-play do not vary substantially. Colyseus is designed primarily with FPS games in mind because we believe FPS game-play is the most difficult to support in a distributed setting. However, we discuss (Section 7.3) how games with different workloads might change our results.

2.4 Distributed Deployment Opportunities

Research designs [27], middle-ware layers [5, 17] and some commercial games [24] have used server clusters to improve the scaling of server-oriented designs. While this approach is attractive for publishers requiring tight administrative control, a widely distributed game deployment can address the scaling challenges and eliminate possible failure modes. In addition, a distributed design can make use of existing third party federated server deployments that we describe below, which is a significant advantage for small publishers.

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 – e.g., "clan" organizations formed by players. Figure 3(a) shows the number
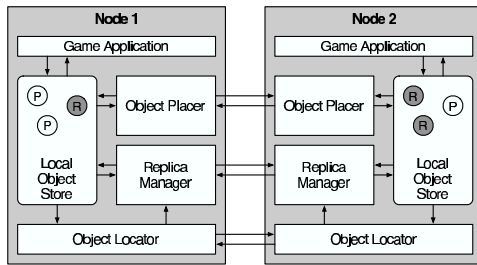
Figure 4: Colyseus components: Circled R's represent secondary replicas, circled P's represent primary objects.

of active third-party servers we observed for several different games. 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 maximum allowed on the server. For most games, 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. Nonetheless, such a widely distributed deployment must address unique problems, such as inter-node communication costs and latencies.

3 Colyseus Architecture

Now, we present an overview of Colyseus, which primarily acts as a game object manager. There are two types of game objects: immutable and mutable. We assume that immutable objects (e.g., map geometry, game code, and graphics) are globally replicated (i.e., every node in the system has a copy) since they are updated very infrequently, if at all. Per-node storage requirements for Quake II and Quake III are about 500MB, though the vast majority of data is for graphics content, which could be elided on game servers. 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, mutable objects and associated think functions are divided amongst participating nodes. Instead of running a single synchronous execution loop, in Colyseus, nodes run separate execution loops in parallel. Figure 4 shows the components in Colyseus that manage objects on each node, which we detail below.

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. In addition to the primary copy, each node in the system may create a secondary replica (or *replica*, for short). These replicas enable remote nodes to execute code that ac-

cesses 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. Section 5 details the synchronization process and the consistency it affords game applications.

In summary, each node has a *local object store* which is a collection of primaries and replicas, a *replica manager* that synchronizes primary and secondary replicas, and a *object placer* which decides where to place and migrate primary replicas. For the purposes of this paper, we assume that objects are placed on the nodes closest to their controlling players, which is likely optimal for minimizing interactive latency, and defer details of the object placer and more sophisticated placement strategies to future work.

Execution Partitioning: Recall that existing games execute a discrete event loop that calls the think function of each object in the game once per frame. Colyseus retains the same basic design, except for one essential 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, rarely will one require access to *all* objects simultaneously 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 gameplay deadlines. Instead, each primary object predicts the set of objects that it expects to read or write in the near future, and Colyseus pre-fetches replicas of these objects. This prediction 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., $10 < x < 50 \land 30 < y < 100$). 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 can use either a traditional randomized DHT or a *range-queriable* DHT as its *object locator*. Range-queries describing area-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*. A subscription and its matching publications are routed to the same location in the DHT, allowing the *rendezvous* node at which they meet to send all publications to their interested subscribers. Since nodes join the object location substrate in a fully self-organizing fashion, so there is no centralized coordination or dedicated infrastructure required in Colyseus.

A particular challenge in applying a DHT to object location in a real time setting is overcoming the delay between the submission of a subscription and the reception of matching publications. Section 6 details two methods to hide object location delays from the game application, and describes the trade-off between locality, dynamics, and complexity when using either DHT substrate in the context of locating game objects.

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 5 shows the primary methods of 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). A few additional interface methods exist for optimizations and are described in subsequent sections.

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). However, the problem of distributing these components is orthogonal to distributing gameplay and is readily addressed by other research initiatives [8, 9].

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' design. In Section 7, we apply Colyseus in a distributed version of Quake II, demonstrating that our observations apply to an existing game.

4.1 Model Workload

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

(a)

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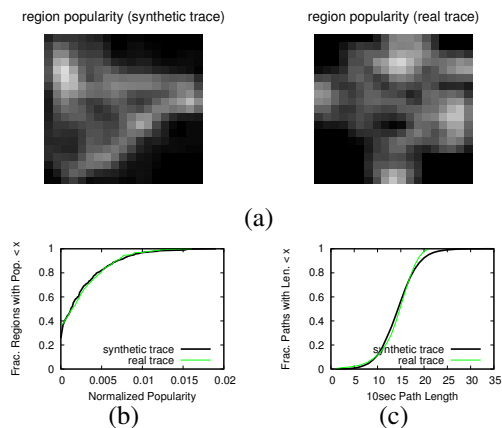
Figure 6: Comparison of our synthetic game trace with a real Quake III trace measured with human players.

plemented as a real game played by bots that runs on top of Colyseus. The game uses synthetic maps with randomly generated obstacles and bots move using a obstacle-sensitive mobility model based on Voronoi diagrams [18]. Mobility parameters 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. Tan, et al. [29] concurrently developed a similar FPS mobility model (without fight logic) and found that it predicted client bandwidth and interest management accuracy well.

4.2 Experimental Setup

We emulate the network environment by running several virtual servers on 5-50 physical machines on Emulab [31]. The environment does not constrain link capacity, but emulates end-to-end latencies (by delaying packets) using measured pairwise Internet latencies sampled from the MIT King dataset [21]. Median round trip latencies for samples are between 80ms-90ms. Due to limited resources and to avoid kernel scheduling artifacts, when running several virtual servers on the same

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
GetVelocity(Vector* dir)	Obtain object's current velocity
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 delta-encoding
UnpackUpdate(Packet* packet, BitMask mask)	Unmarshall an update for this object

Figure 5: The interface that game objects implement in applications running on Colyseus.

physical machine, we artificially dilate time (e.g., using a dilation factor of 3, 1 experimental minute lasts 3 actual minutes) by dilating all inter-node latencies, timers, and timeouts accordingly. Hence, our latency results do not include computational delays, but since our configurations emulated at most 8 players per server, computational delay would be negligible even in a real game (e.g., see Figure 1(a)). In addition, UDP is used for transport, so the impact time dilation would have on TCP does not affect our results. Each game/experiment run lasts 8 minutes, which is about half the time of a typical FPS game round.

Different experiments vary two main parameters: players-per-node and map-type. We use two player-per-node counts: 1 player per node, which we call the peer-to-peer scenario (p2p), and 8 players per node, which we call the federated server scenario (fed). We use the p2p scenario to illustrate scaling behavior since it allows us to run the most virtual nodes per physical node in our testbed. Similarly, we use the fed scenario when quantifying the characteristics of a particular configuration, since it allows us to run the most total players in the game world, increasing interactivity. In general, increasing the number of players per node (while average density remains constant) increases communication costs linearly (since all players are randomly spawned in the map) and does not substantially affect the other metrics we measure (which are mostly functions of node count). We have validated these properties in most of our experiments.

We evaluate two types of maps: square (sqr) and rectangular (rect). In both types, we select the map area that achieves the same *average* player density as in a full Quake III game although the density *distribution* follows the Zipf-like model we observed. The height of rect maps is always equal to the diameter of a 16 player Quake III map, while sqr maps vary both dimensions equally. rect maps simulate a *linearization* (e.g., using Hilbert space-filling curves [26]) of a multi-dimensional map, which may be useful in some games where not much locality is sacrificed. Note that although the maps we use are uniform shapes, the area that is traversed during game-play obeys actual non-uniform characteristics, as demonstrated by Figure 6(a). The map type primarily impacts the performance of the object location layer,

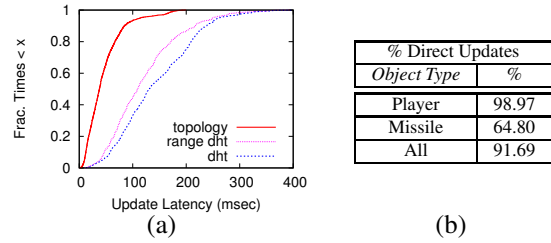| % Direct Updates	
Object Type	*%*
Player	98.97
Missile	64.80
All	91.69

(a) (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.

because the regions of each type will have different locality properties when mapped onto the DHT identifier space.

In the following sections, we describe the details of the replica manager and object locator, using the above setup to quantify important points. Experiments are named using the convention $node\_count$-{p2p,fed}-{sqr,rect} to indicate their configurations.

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. In our current implementation, primaries synchronize replicas in an identical fashion to how dedicated game servers synchronize clients: each frame, if the primary object is modified, a delta-encoded update is shipped to all replicas. Similarly, when a secondary replica is modified, a delta-encoded update is shipped to the primary for serialization. Although other update models are possible for games on Colyseus, this model is simple and reflects the same loose consistency in existing client-server architectures.

Decoupling Location and Synchronization: An important aspect of Colyseus' replica manager is the decoupling of object discovery and replica synchronization. Once a node discovers a replica it is interested in, it synchronizes the replica directly with the primary from that point on. The node periodically registers interest with the node hosting the primary to keep receiving updates to the replica.

Another strategy would be to always place each ob-

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.

ject on the node responsible for its region (as in *cell*-based architectures [17, 20, 24]). However, FPS game workloads exhibit rapid player movement between cells, which entails migration between servers. For example, in a 96-fed-rect game with one region per server, this approach causes each player to migrate once every 10 seconds, on average, and hence requires a frequency of connection hand-offs that would be disruptive to game-play. Yet another design would be to route updates to interested parties via the rendezvous node in the DHT (as in [20]). However, this approach adds at least one extra hop for each update.

To quantify the impact of decoupling, Figure 7(a) compares the one-way direct latencies between 96 nodes in a real world end-host topology [21] (topology) and the delivery latency of publications and subscriptions in a 96-fed-rect experiment using both a range-queriable DHT (rangedht) and a traditional DHT (dht). Although routing through either substrate achieves much better than $\log n$ hops due to the effectiveness of route caching with a highly localized workload, the delays are still significantly worse than sending updates directly point-to-point, especially considering the target latency of 50-100ms in FPS games [1].

In Colyseus, the only time a node incurs the DHT latency is when it must discover 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 game if each player were 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, more aggressive interest *prediction*, which we discuss in the next section, 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.

Proactive Replication: To locate short-lived objects like missiles faster, Colyseus leverages the observation

that most objects originate at locations close to their creator, so nodes interested in the creator will probably 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 (via an optional AttachTo() method that adds to the object API in Figure 5). 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). We see that in practice, this simple addition improves consistency of missiles significantly. For example, in a 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. In addition, we note that the overhead is negligible.

Replica Consistency: In Colyseus, writes to replicas are tentative and are sent to the primary for serialization. Our model game applies tentative writes (tentatively), but a different game may choose to wait for the primary to apply it. In other words, individual objects follow a simple primary-backup model with optimistic consistency. The backup replica state trails the primary by a small time window ($\frac{1}{2}$ RTT, or, from the results shown in Figure 7(a), <100ms for 93% of node pairs), and are *eventually consistent* after this time window.

In addition to per-object consistency, it is desirable to consider *view* consistency in the context of a game. The *view* of a server (or a player) is the collection of objects that are currently within the union of the server's (player's) subscriptions. Here, we discuss view consistency with respect to the TACT model [32], since its continuous range of consistency/performance trade-offs likely to be most useful to game applications. In the TACT model, the view of a server can define a *conit*, or unit of consistency. There are two types of view inconsistency in Colyseus: first, a server is missing replicas for objects that are within its view; and second, replicas that are within its view are missing updates or have updates applied out-of-order. Both types of inconsistency actually exist in *any* application using the TACT model, since when a new conit is defined, time is required to first replicate the desired parts of the database to "initialize" the conit (resulting in the first type) before maintaining it (which can result in the second type). The first type is

simply exacerbated in a distributed game because views change frequently and reads often can not wait for views to finish forming.

Since Colyseus introduces missing replicas as a significant source of inconsistency, we use the number of missing replicas as the primary metric when evaluating consistency. Inconsistency due to missing or late updates can be managed in an application specific manner using the TACT model (with game specified bounds on order, numerical, and staleness error). Hence, Colyseus is flexible enough to support games with different view consistency requirements.

We believe that most fast-paced games would rather endure temporary inconsistency rather than have the affects of writes (i.e., player actions) delayed, so our implementation adopts an optimistic consistency model with no bounds on order or numerical error in order to limit staleness as much as possible. As described above, this ensures replica staleness remains below 100ms almost all of the time. Limited staleness is usually tolerable in games since there is a fundamental limit to human perception at short time-scales and game clients can extrapolate or interpolate object changes to present players with a smooth view of the game [1]. Moreover, we observed that frequently occurring conflicts can be resolved transparently. For example, in our distributed Quake II implementation, the only frequent conflict that affects game-play is a failure to detect collisions between solid object on different nodes, which we resolve using a simple "move-backward" conflict resolution strategy when two objects are "stuck together." The game application can detect and resolve these conflicts before executing each frame.

6 Locating Distributed Objects

To locate objects, Colyseus implements a distributed location service on a DHT. Unlike other publish-subscribe services built on DHTs [6], the object locator in Colyseus must be able to locate objects using range queries rather than exact matches. 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 [28, 25] 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 loca-

tor 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 [3, 19] may be 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 value in the range. This node then forwards the query to other nodes in the range. 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 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 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 well). Colyseus expands a the bounding volume subscribed to by an object (via `GetInterest()`) using the following formula:

$$Vol.min- = PredTime \times PredMoveUpLeft + PubTime$$
$$Vol.max+ = PredTime \times PredMoveDownRight + PubTime$$

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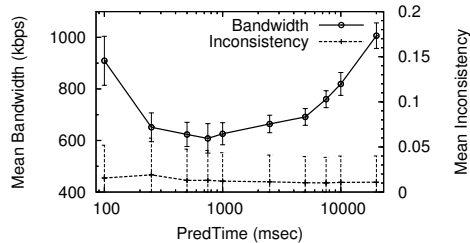
Figure 8: The impact of varying PredTime **on total mean node bandwidth and local object store inconsistency.**

(PredTime), which a per-object configuration parameter. The default implementation uses moving average of an object's velocity to estimate PredMoveUpLeft and PredMoveDownRight, but the application can override it (via an additional object API method) 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. Thus, if predicted subscriptions are stored in the DHT with a TTL = PredTime, it is unlikely they will have to be refreshed within that time. Subscription prediction amounts to *speculative pre-fetching* of object location attributes.

Speculation can incur overhead. Figure 8(a) shows the impact of tuning subscription prediction (by varying PredTime) in a 50-fed-rect game. The top line plots the total mean bandwidth required by each node, while the bottom line shows the mean local object store inconsistency, defined as 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 is not yet discovered). 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. Extraneous delivery of matched publications does not result in unnecessary replication, since upon reception of a pre-fetched publication, a node will 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* subscriptions locally. Hence, overhead is solely due to extra received publications. 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. Hence, we can automatically optimize PredTime without affecting the level of inconsistency observed by the game. In addition, although we focused on using prediction to minimize communication overhead, we can also trade-off overhead for improved consistency by increasing PubTime.

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 the DHT), 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 [15].

Soft State Storage: In most publish-subscribe systems, only subscriptions are registered and maintained in the DHT while publications are not. 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 of possible staleness (i.e., we add PubTTL×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: a traditional DHT (dht) versus a range-queriable DHT (rangedht). In general, our results show rangedht incurs lower bandwidth overhead compared to a dht by utilizing contiguity in data placement and has good scaling properties if the game map can be linearized.
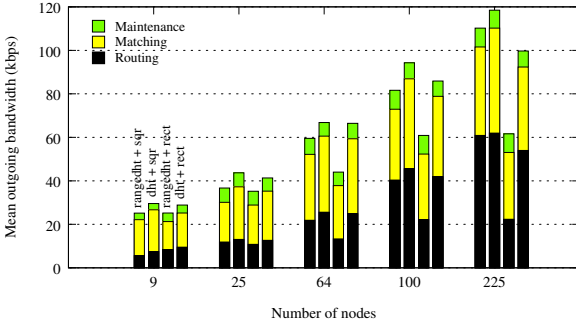
Figure 9: Scaling of per-node bandwidth using a dht **and** rangedht **in a** p2p **game with** sqr **and** rect **maps.**

rangedht incurs higher object discovery latency compared to a dht, but at time scales of 100ms, the resultant inconsistency in game-state is indistinguishable. Finally, rangedht more fairly balances object location overhead between all nodes, suggesting that it is more suitable in bandwidth constrained deployments.

Colyseus uses an implementation of Mercury [3] with the extensions described earlier in this section. Mercury is used both as the dht and the rangedht, 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 rangedht, the x dimension is used as the key attribute. In both cases, each node caches $2 \log(n)$ recently used routes.

6.3.1 Communication Costs

Figure 9 compares the average per-node outbound bandwidth requirements for object discovery, varying the number of nodes and the map type in p2p games. 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 three components: sending and routing publications and subscriptions in Mercury (routing), delivering matched publications and subscriptions (matching), and DHT maintenance (maintenance). In all cases, rangedht consumes less bandwidth than dht.

Performance of a dht is similar for both sqr and rect maps. However, a rangedht performs noticeably better with rect maps because the total span of the key-space is larger relative to the width of subscriptions, so each subscription covers fewer nodes.

Scaling Behavior: Since the map area grows linearly with the number of players and subscription area is constant, as more nodes are added to a rangedht, the number of nodes contacted for each subscription stays constant if using a rect map, but grows proportional to $\sqrt{n}$ if using a sqr map. For a dht substrate, this number stays constant irrespective of the map type. However, the lack of locality in the generated subscriptions results in higher rout-

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

Table 2: Effectiveness of a load-balanced rangedht**. The percentage of missing replicas shows the mean and standard deviation.**

ing overhead since caching routes becomes less effective. In addition to these effects, since player interaction grows as the number of players in the game increase, the overall matching traffic also grows (as Figure 9 shows). Hence, we observe that both dht and rangedht routing bandwidth scale poorly using sqr maps, but rangedht scales well with a linearized rect map.
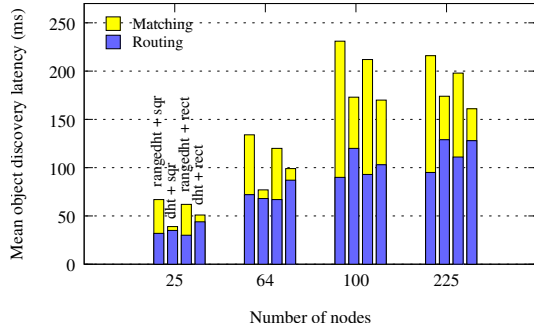
Load Balancing: Since popularity of the regions in the model workload is Zipfian, nodes in the routing ring responsible for such regions can get considerably more traffic than others. We now focus on the effectiveness of the leave-join load-balancing mechanisms built into the Mercury rangedht, which dynamically moves lightly loaded nodes to heavily loaded regions the DHT. The number of publications and subscriptions routed per second, averaged over a 30-second moving window, is used as the measure of the load.

Table 2 compares the bandwidth and view inconsistency (see Section 6.3.2) for a 96-fed-rect game. We find that a rangedht with load balancing enabled (loadbal) reduces the maximum per-node bandwidth by about 25% (relative to the mean) and the maximum per-node matching bandwidth by about 42%, compared to a dht. While partitioning a busy range may not necessarily result in decreasing routing load (since each subscription will have to visit all nodes that span its range), 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 is not substantially higher, suggesting that players do not lose many updates due to the leave-join dynamics of load balancing.

6.3.2 Latency and Inconsistency

In this section, we evaluate the impact of the routing substrate on game-state consistency. We first evaluate how long it takes for a node to discover and replicate an object that it is interested in, which provides an estimate of the worst case delay that a view might have to endure. We then examine the impact that this 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 the different DHTs and map types. This latency is broken down into two

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

Figure 11: The fraction of replicas missing averaged across all time instances as we scale the number of servers.
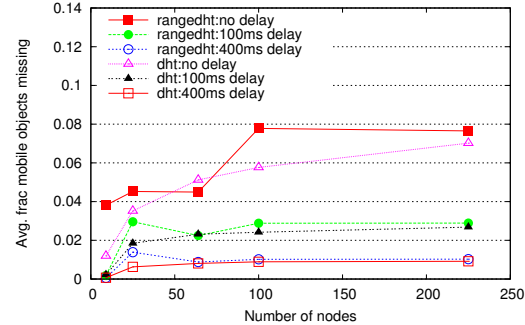
parts: routing the subscription to the *first* (left-most) rendezvous (routing), and delay incurred at the rendezvous before a matching publication arrives (matching). To completely construct a replica, an additional delay of 1.5 RTT (135ms on average) must be added: 0.5 RTT for delivering the publication, and 1 RTT for fetching the replica. However, this latency is independent of the location substrate.

The routing delay for subscriptions scales similarly in both DHTs, as expected. Both are able to exploit caching so the median hop count is at most 3 in all cases. However, the matching latency is higher for the rangedht case. This is because the matching component incorporates the latency incurred when spreading the subscription *after* reaching the left-most rendezvous point. Thus, dht incurs bandwidth overhead by sending multiple disjoint subscriptions, but obtains an small overall latency advantage.

Discovery latencies are only incurred when an interesting object is first discovered (e.g., when a player enters a new room or an object enters 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 is less than 100ms for most node pairs (see Section 5). Incorporating proximity routing techniques [14] into our Mercury implementation can further reduce the latency of the routing component in both cases.

View Inconsistency: Now we examine the impact of discovery delay on view consistency. We define view consistency as the ratio of replicas missing and total replicas in a node's subscriptions (summed over all game frames). Figure 11 shows the fraction of replicas missing for a dht and rangedht in p2p-sqr games, if we allow 0ms, 100ms, and 400ms to elapse after the objects enter a node's subscriptions.

We see that inconsistency in game state is approximately the same irrespective of the choice of the routing substrate. The rangedht has slightly higher inconsis-

tency due to the higher object discovery latency. However, this difference vanishes if we allow for a small delay of 100ms. For both DHTs, 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 1% missing if we allow for a 400ms delay (4 frames).

7 Evaluation With a Real Game

To demonstrate the practicality of our system, we modified Quake II to use Colyseus. 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 use a large, custom map with computer controlled bots as the workload, and the same Emulab testbed setup described in section 4 for our Quake II evaluation. However, we did *not* artificially dilate time, so all numbers reported take into account actual execution times. We use the Mercury rangedht as the object location substrate, and linearize the game map when mapping it onto the DHT. Further details about our Quake II prototype and additional results can be found in an associated technical report [4].

7.1 Communication Cost

Figure 12 compares the bandwidth scaling of Colyseus running p2p games with the client-server and broadcast architecture alternatives. We simulate the alternatives

| (a) Model Workload | (b) Quake II | (c) Quake II (Same Density) |
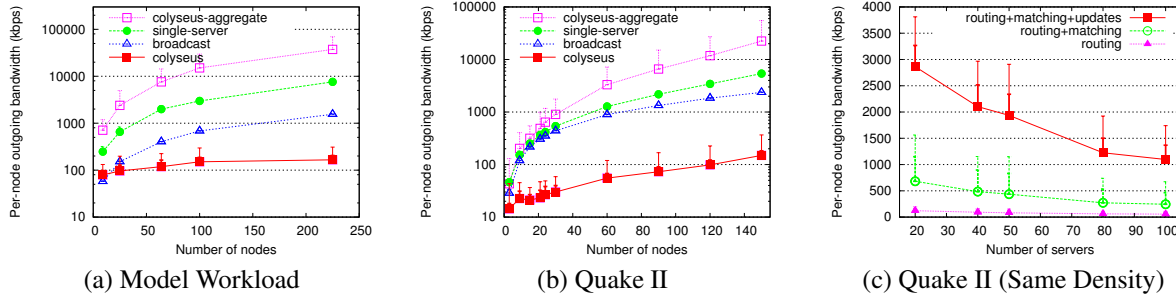
Figure 12: Bandwidth scaling properties using (a) the model workload and (b) the Quake II workload (note the logarithmic scale). Part (c) shows the scaling of Quake II, with a constant number of players.

using the same game-play events as the real execution on Colyseus.

Figure 12(a) shows the scaling properties with rect maps under our model workload. The workload keeps mean 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 colyseus-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' 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 for networks with sufficient capacity.

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 map, regardless of the number of players. Thus, the average density of players increased with the number of 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' 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 may not reduce per-node bandwidth cost by the same factor. In this experiment, we see a 3-fold decrease in communication cost per node with a 5-fold increase in the number of

nodes, so overhead is less than a factor of 2. We expect similar bandwidth scaling characteristics to hold for our model workload and Quake II if average player density were fixed. This result shows that the addition of resources in a federated deployment scenario can effectively reduce per-node costs.

If we hand-tune update delta sizes so they were smaller, the client-server and broadcast architectures would perform better. However, Figure 12(c) shows that updates also account for over 75% of Colyseus' costs, so Colyseus would get a substantial benefit as well. Moreover, the scaling properties would not change.

7.2 View Inconsistency

We now examine the view inconsistency, i.e., fraction of missing local replicas, observed in the Quake II workload (Section 6.3.2 showed this for the model workload.) Figure 13(a) shows the fraction of replicas missing as we scale the number of nodes for a p2p scenario. The results are very similar to those obtained with the model workload. Note that nearly one half of the replicas a node is missing at any given time instance arrive 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-fed game. 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 for a replica to arrive and to over 80% of the time if we wait 400ms for a replica to arrive. The inconsistency is less for sparser game playouts.

Although the fraction of missing replicas is low, objects in a view can differ in semantic value; e.g., it is probably more important to promptly replicate a missile that is about to kill a player than a more distant object. In general, a game-specific inconsistency metric might consider the type, location, and state of missing objects to reflect the total impact on game-play quality. Due to locality in object movement, Colyseus' replication model accounts for at least one important aspect: location. Figure 13(c) compares the distance (over time)
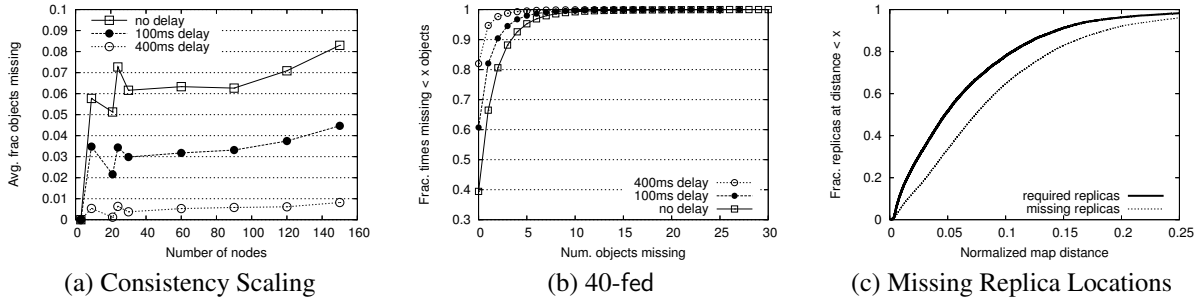
(a) Consistency Scaling   (b) 40-fed   (c) Missing Replica Locations

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-fed **game. (c) CDF showing the distance of missing replicas from a subscriber's origin.**

of a player to objects in its area-of-interest and the distance to those that are missing. Replicas that are missing from a view tend to be closer to the periphery of object subscriptions (and hence, farther away from the subscriber and probably less important). The difference in the distributions is not larger because subscription sizes in Quake II are variable, so objects at the periphery of a subscription may still be close to a player if they are in a small room. We leave a more game-play-centric evaluation of view inconsistency to future work.

7.3 Discussion

Throughout our evaluation of Colyseus we have used workloads derived from Quake II or Quake III, which we believe are representative of FPS games in general. However, questions remain about how representative our results are to other game genres, such as massively multiplayer Role Playing Games (RPGs.)

RPGs have lower update rates and have much smaller per-player bandwidth requirements than FPS games [7]. Hence, they are usually designed to tolerate much longer delays in processing player actions. In general, these characteristics imply that an RPG game implemented on Colyseus would incur lower communication costs than what we have measured. We do not expect discovery delay and replica staleness to change substantially because they are primarily functions of system size and network topology. Consistency may actually improve since players generally move slower in RPG games, and players have a higher tolerance for inconsistency (lower update rates imply existing game clients already tolerate staler state.) Thus, although we have demonstrated two case studies that effectively used Colyseus, we believe it can also be applied to less demanding game genres.

8 Related Work

There are a number of other commercial and research game architectures. Some games (e.g., MiMaze [12] and most Real Time Strategy (RTS) games [2]) use *parallel simulation*, where each player simulates the entire game world. All objects are globally replicated and kept

consistent using lock-step synchronization and update broadcast, resulting in quadratic scaling behavior and limiting response time to the speed of the slowest client. These deficiencies are tolerated in RTS games because they rarely involve more than 8 players.

Second-Life [24] and Butterfly.net [17] perform interest filtering by partitioning the game world into disjoint regions called *cells*. SimMUD [20] makes this approach fully distributed by assigning cells to keys in a DHT, though, unlike Colyseus, primaries in SimMUD reside on the rendezvous node. Although these approaches share some traits with Colyseus, we believe that we are the first to demonstrate the feasibility of implementing a real-world game on a distributed architecture that is (1) not designed for a centralized cluster ([24, 17]), and (2) that supports FPS games, which have much tighter latency constraints than RPGs (which were targeted by SimMUD). Furthermore, using a cell-based design with an FPS game can result in frequent object migration, as shown in Section 5.

Several architectures proposed for Distributed Virtual Reality environments and distributed simulation (notably, DIVE [11], MASSIVE [13], and High Level Architecture (HLA) [16]) 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 hundreds of participants without the use of IP multicast.

9 Summary and Future Work

This paper described the design, implementation and evaluation of a distributed architecture for online multiplayer games. Colyseus enables low-latency game-play via three important design choices: (1) decoupling object discovery and replica synchronization, (2) proactive replication for short-lived objects, and (3) pre-fetching of relevant objects using interest prediction. Our investigation showed that a range-queriable DHT achieves bet-

ter scalability and load balance than a traditional DHT when used as a object location substrate, with a small consistency penalty. We believe our adaptation of a commercial game (Quake II) demonstrates the practicality of Colyseus' design.

Nonetheless, our work on Colyseus is on-going. For example, Colyseus enables three new avenues for cheating: (1) nodes can modify objects in their local store in violation of game-play logic (2) nodes can withhold publications or updates of objects they own, and (3) nodes can subscribe to regions of the world that they should not "see." Although our work on addressing cheating is nascent, we believe we can leverage Colyseus' flexibility in object placement by carefully selecting the owners of primary objects to limit the damage inflicted by malicious nodes. Moreover, nodes holding replicas can act as *witnesses* to detect violations of game-play rules.

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

10   Acknowledgements

References

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

[2] BETTNER, P., AND TERRANO, M. 1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond. *Gamasutra* (Mar. 2001).

[3] BHARAMBE, A., ET AL. Mercury: Supporting scalable multi-attribute range queries. In *SIGCOMM* (Aug. 2004).

[4] BHARAMBE, A., ET AL. A Distributed Architecture for Interactive Multiplayer Games. Tech. Rep. CMU-CS-05-112, CMU, Jan. 2005.

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

[6] CASTRO M., ET AL. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE J. on Sel. Areas in Comm. 20*, 8 (Oct. 2002).

[7] CHEN, K., ET AL. Game Traffic Analysis: An MMORPG Perspective. In *NOSSDAV* (June 2005).

[8] DABEK, F. ET AL. Wide-area cooperative storage with CFS. In *SOSP* (Oct. 2001).

[9] DRUSCHEL, P., AND ROWSTRON, A. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP* (Oct. 2001).

[10] FENG, W., ET AL. Provisioning on-line games: A traffic analysis of a busy counter-strike server. In *IMW* (Nov. 2002).

[11] FRÉCON, E., AND STENIUS, M. DIVE: A scaleable network architecture for distributed virtual environments. *Dist. Sys. Eng. J. 5*, 3 (1998), 91–100.

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

[13] GREENHALGH, C., ET AL. Massive: a distributed virtual reality system incorporating spatial trading. In *ICDCS* (June 1995).

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

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

[16] IEEE standard for modeling and simulation high level architecture (HLA), Sept. 2000. IEEE Std 1516-2000.

[17] IBM and Butterfly to run PlayStation 2 games on Grid. `http://www-1.ibm.com/grid/announce_227.shtml`, Feb. 2003.

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

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

[20] KNUTSSON, B. ET AL. Peer-to-peer support for massively multiplayer games. In *INFOCOM* (July 2004).

[21] MIT King Data. `http://www.pdos.lcs.mit.edu/p2psim/kingdata`.

[22] Quake II. `http://www.idsoftware.com/games/quake/quake2`.

[23] Quake III Arena. `http://www.idsoftware.com/games/quake/quake3-arena`.

[24] ROSEDALE, P., AND ONDREJKA, C. Enabling Player-Created Online Worlds with Grid Computing and Streaming. *Gamasutra* (Sept. 2003).

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

[26] SAGAN, H. *Space-Filling Curves*. Springer-Verlag, New York, NY, 1994.

[27] SHAIKH, A., ET AL. Implementation of a Service Platform for Online Games. In *NetGames* (Aug. 2004).

[28] STOICA, I., ET AL. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM* (Aug. 2001).

[29] TAN, S. A., ET AL. Networked game mobility model for first-person-shooter games. In *NetGames* (Oct. 2005).

[30] Torque Networking Library. `http://www.opentnl.org`.

[31] WHITE, B., ET AL. An integrated experimental environment for distributed systems and networks. In *OSDI* (Dec. 2002).

[32] YU, H., AND VAHDAT, A. Design and Evaluation of a Conit-based Continuous Consistency Model for Replicated Services. *ACM Trans. on Comp. Sys.* (Aug. 2002).