A Distributed Architecture for Interactive Multiplayer Games

Ashwin R. Bharambe **Jeff Pang**
Srinivasan Seshan

January 2005
CMU-CS-05-112

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper presents the design, implementation, and evaluation of *Colyseus*, a distributed architecture for interactive multiplayer games. Colyseus distributes dynamic game-play state and computation while adhering to tight latency constraints and maintaining scalable communication costs. Two key architectural decisions form the basis of our design: first, a *single copy consistency model* for game objects allows low-latency read/writes at the cost of weaker consistency, which is tolerated by most multiplayer games. Second, we utilize locality and predictability in the game workload to speculatively and quickly *pre-fetch* objects needed for performing game logic computation. We have implemented Colyseus and demonstrated its practicality by modifying a popular first person shooter (FPS) game called Quake II to use Colyseus for distributing game state across multiple servers or peers. While current single server implementations can support many tens of players, our playable-prototype shows that Colyseus is easily able to support low-latency game-play for hundreds of participants. In addition, our results show that per-node bandwidth requirements for Colyseus are an order of magnitude lower than traditional client-server or broadcast architectures in two different deployment scenarios.

This research is supported in part by the Pittsburgh Digital Greenhouse grant PDG-A002843.

Keywords: Distributed applications, multi-player games, range queries, peer-to-peer systems, quake

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 other active players. This design suffers from the well known robustness and scalability problems of single server designs. For example, complex game-play and AI computation 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 distributed applications is difficult. The most fundamental challenge in distributing an application is in 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 the 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 such 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 of a player occur upon objects which are physically close to the player.

The challenge, then, is to arrive at a scalable and efficient state and logic partitioning that enables reasonably consistent game-play without incurring much latency. This paper presents the design, implementation and evaluation of Colyseus, a novel distributed architecture for interactive multiplayer games designed to achieve the above goals. Our design is based on two key architectural decisions: First, distributed objects in Colyseus follow a *single-copy consistency model* – i.e., all writes to an object are serialized through exactly one node in the system. This allows low-latency reads and writes at the cost of weak consistency. This mirrors the consistency model of a client-server architecture, albeit on a per-object basis. Secondly, Colyseus utilizes locality and predictability in the movement patterns of players to *speculatively pre-fetch* objects needed for driving game logic computation. Efficiently locating objects to pre-fetch is achieved using a scalable distributed range-query lookup service.

Our design 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. This concrete case study illustrates the simplicity of using our architecture to distribute existing game implementations. In our distributed Quake II implementation, unmodified Quake II clients can connect to any instance of the Colyseus-based distributed Quake II 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 infrastructure (with clients connecting to the closest distributed server). Our measurement of this prototype on an Emulab testbed indicates that Colyseus scales to hundreds of players by distributing game traffic well 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.

The rest of the paper is organized as follows. Section 2 provides background about game design, game deployment and scaling properties. Section 3 provides an overview of the architecture of Colyseus, while Sections 4 and 5 detail the object location and replica management components. Section 6 presents our evaluation of Colyseus. Section 7 surveys related work. Finally, Section 8 concludes and discusses future work.

2 Background and Motivation

In this section, we survey the requirements of online multiplayer games in detail 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 relatively large *game world* (a two or three dimensional space where the game is played). This description applies to a large number of 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 [22, 23].

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, buildings, walls, players' avatars, computer controlled characters, doors, 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.

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.

[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.

2.2 Client-Server Scaling Properties

The most significant drawback of this client-server game architecture is its reliance on a single server to maintain the game. 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: (1) the number of objects in play (*NumObjs*), (2) the average size of those objects (*ObjSize*), and (3) the game's frame-rate (*UpdateFreq*). In Quake II, if we only consider objects representing players (which tend to dominate the game update traffic since they are the most dynamic), *NumObjs* ranges from 8 to 64, *ObjSize* is ∼200 bytes, and *UpdateFreq* is 10 updates per second. A naïve server implementation which simply broadcasts the updates of all objects to all game clients (*NumClients*) would incur an outbound bandwidth cost of *NumClients* × *NumObjs* × *ObjSize* × *UpdateFreq*, or 1-66Mbps in the case of Quake II games between 8 and 64 players.

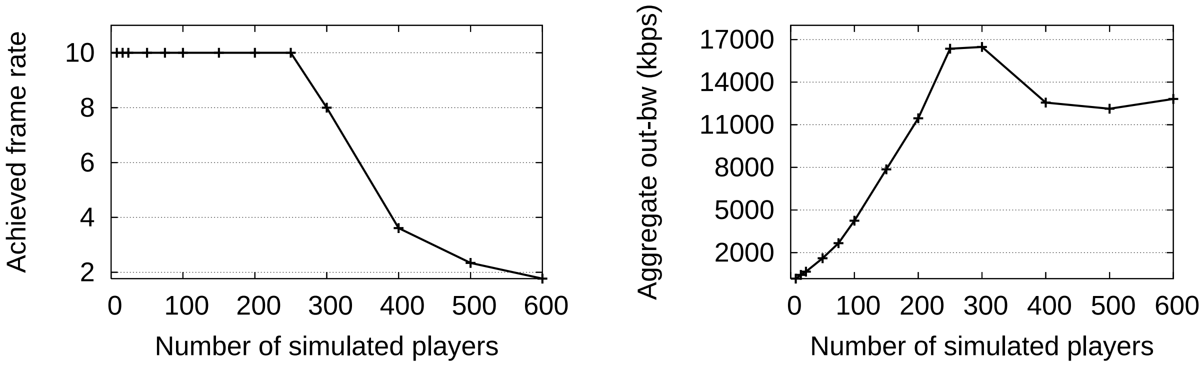
Two common optimizations used by games are *area-of-interest* filtering and *delta-encoding*.[2] Individual players typically only interact with or see a small portion of the game world at any one time. Servers need only update clients about objects in this area-of-interest, thus, reducing the number of objects transferred from the total set of objects (*NumObjs*) to the number of objects in this area (*NumAoiObjs*). If we only consider players, *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 (*ObjSize*) to the average delta size (*AvgUpdateSize*), which is about 24 bytes in Quake II. Thus, the optimized outbound server bandwidth cost would be *NumClients* × *NumAoiObjs* × *AvgUpdateSize* × *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 does not rate-limit clients. We run the game for a 10 minute duration 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[3] 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

[2] A third mechanism sometimes used is to rate-limit update traffic by dropping "less important" updates.

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



(a) (b)

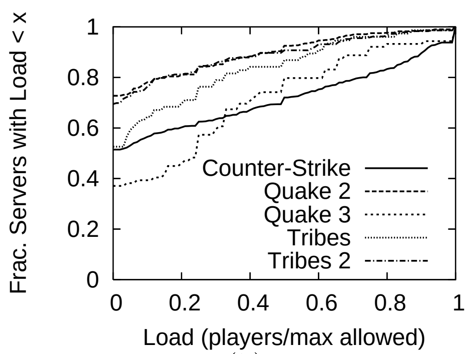
Figure 1: Computational and network load scaling behavior at the server end of client-server system.

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 Distributed Deployment Opportunities

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

(a)



(b)

Figure 2: (a) Observed number of third-party deployed servers for several games, (b) Load on these servers.

Research designs [26], 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 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 (such as "clan" organizations formed by players). Figure 2(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 2(b) plots the cumulative distribution of load on the different sets of servers, where we define load 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. Other efforts [13, 18] target purely peer-to-peer deployments. Note that centrally administered, distributed server infrastructure, such as Akamai, could leverage many of the designs that we discuss in this paper. Server clusters could also make use of our designs to distribute computation among the cluster nodes. However, our design must still address problems, such as high communication costs and inter-node latencies, that are not present in clusters.

3 Architecture Overview

We now present an overview of Colyseus, the general architecture we have developed for distributed multiplayer games. In our game model, we consider two classes of game state: immutable and mutable. Immutable state (e.g., map geometry, game code, and graphics) is globally replicated, i.e., every node in the system has a copy. We refer to the collection of mutable objects (e.g., players' avatars, computer controlled characters, doors, items) as the *global object store*. In the rest of the paper, we do not focus on immutable state since it is updated very infrequently, if at all.

Our architecture is an extension of the Quake II model described earlier in Section 2.1. However, in order to adapt the latter for a distributed setting, we need to solve a number of problems: first, mutable state must be partitioned in some manner amongst participating nodes. Similarly, the think functions associated with objects must also be partitioned for execution. Secondly, recall that each game instance runs a synchronous event execution loop. A distributed system inherently introduces asynchronous *parallel* execution loops, one on each node. We need a mechanism to deal with the inconsistency that can result due to write-sharing arising from such parallel executions. Finally, if possible, it is desirable to place objects in an optimal manner to reduce communication costs. The rest of this section addresses the above aspects in greater detail.

State Partitioning

We require that each object in the system is identified by a globally unique identifier which we call a `GUID` (e.g., a large pseudo-random number). Our implementation uses 80-bit `GUID`s) and each node is uniquely identified by a routable End-point Identifier or `EID`.

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 must be transmitted to the primary owner. This owner node provides a serialization order to updates. By using a single owner, we avoid incurring delays associated with running a quorum protocol between multiple writers. In addition to the primary copies, each node in the system may create secondary replicas (or *replicas*,

[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.

for short) of objects in the system. These replicas are created in order to enable the execution of game code on each node. Replicas are weakly consistent copies of the primary that are loosely synchronized with the primary in an application dependent manner. Modifications made to a secondary replica must be transmitted to the primary to be committed. An application may or may not choose to expose tentative local updates of secondary replicas to clients of a node. 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, execution of existing games is basically a discrete event loop that calls the think function of each object in the game. 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 complete execution of this code, a node must create a secondary replica of the object. Performing this on-demand can result in a stall in game execution, resulting in a violation of the real-time deadlines for game-play. Our solution to this problem is for each primary object to predict the set of objects that it expects to read or write in the near future. Colyseus manages the contents of the local object store and pre-fetches the objects in the read and write set of all primary objects on the node. The prediction of the read and write set for each primary object is specified as a selective filter across the objects attributes. We refer to this filter as the *area-of-interest* of the primary object. We have found that most games can succinctly express their areas-of-interest using range predicates over multiple object attributes. This works 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 its object store.

State Placement

The assignment of primary copies to nodes can have a significant impact on the performance of the system. For example, by clustering primary copies of objects that are within each other's area-of-interest, we can minimize the number of replicas created in the system. By reducing the number of replicas, we, in turn, reduce the amount of network traffic that is used to propagate updates between replicas and primary copies. Another example is that the placement of the primary copies of a player's avatar objects on the server that is closest to the client would minimize interactive latency for the player. In general, we believe that there are a variety of opportunities for optimizing object placement for certain games (e.g., to dynamically balance load, minimize interactive latency, etc.). One key primitive that Colyseus supports to enable this optimization is the reassignment of a primary copy to a different node. This requires transferring the entire state and code associated with an object to a new node. Note that this may require significant overhead and that the primary copy will be unavailable during the transfer. As a result, the optimization benefits must be significant to perform such transfers.

class ColyseusObject	
IsReplica()	Determines if object is replica or primary
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)	Unmarshall an update for this object

Figure 3: The interface that game objects implement in applications running on Colyseus.

We believe that the migration primitive is important for enabling features such as robustness against node departures and cheating prevention, which we have not yet explored.

Example: Consider a player's avatar in Quake II. At any given time, it will be managed by a single node in the system (e.g., the node that the client is connected to). The player's think function will process input commands from the player, modify its position in the game, and other objects the player interacts with. Hence, the node requires the current game map, which is immutable and can be shared by all nodes. In addition, the node must send the current view of the game to the player. As a result, all other objects, including other player avatars, monsters, items, and missiles, in the immediate vicinity of the player need to be present on a node. Moreover, the player's avatar may modify nearby objects; e.g., he might shoot a wounded monster, reducing its health and killing it. If the primary copy of the monster object is located on another node, the modification is sent to this node which determines the commit order of all updates.

Application Interface

From our experience modifying Quake II to use Colyseus (described in Section 6.1) and our examinations of the source code of several other games, we believe that this model is sufficient for implementing the logic for most important game operations. Figure 3 shows the core of the object interface for game objects managed by Colyseus.[5] There are certainly other components of online games that this model does not encompass, such as content distribution (e.g., game patch distribution) and persistent storage (e.g., storing persistent player accounts), but the problem of distributing these components is orthogonal to distributing game-play and is readily addressed by other research initiatives [14, 16].

Colyseus allows nodes to join the system in a fully self-organizing fashion, so there is no centralized coordination required. In addition, we believe it places few requirements on game developers. The only major additions to the centralized object store programming model are that: (1) each object publishes a small number of attributes (such as their location) that others can use to look it up; (2) and each object specifies its area-of-interest with range queries on published attributes (i.e., a declarative variant of how area-of-interest is currently computed). These published attributes can be thought of as *naming* attributes. In Figure 3, these are implemented using the `GetLocation()` and `GetInterest()` object methods.

3.1 Architecture Components

There are three primary challenges in designing an architecture to realize the preceding model:

[5] This excludes optional features used for optimization, described in later sections.

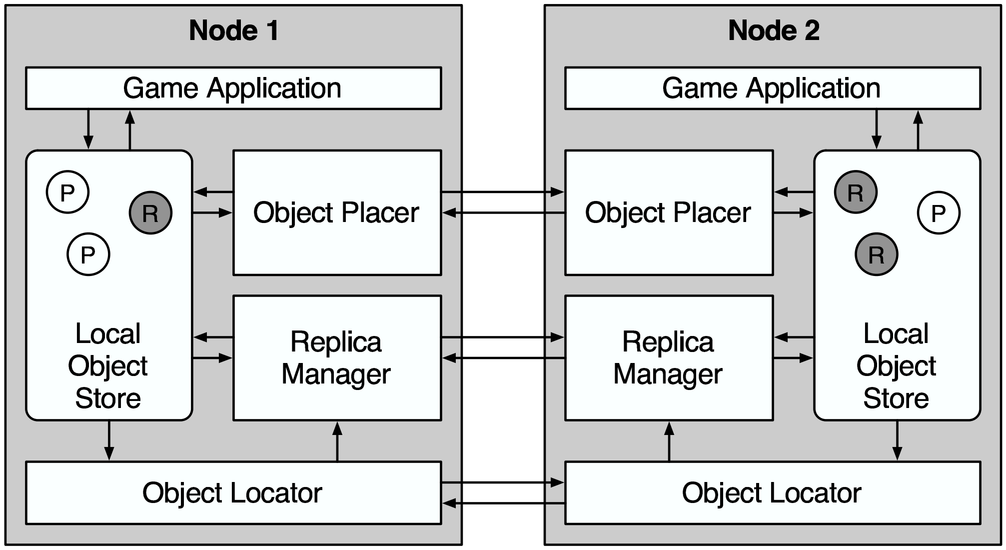


Figure 4: The components of Colyseus. Circled R's represent secondary replicas, circled P's represent primary objects.

Object Location: Given a set of primary objects, a node requires a mechanism to discover all other objects the primaries are interested in.

Replica Management: Once a node has determined remote objects it requires (i.e., objects for which it does not maintain the primary), it must maintain loosely-synchronized replicas of those objects.

Object Placement: Colyseus should (re)assign primary objects to nodes to optimize some metric such as communication cost.

Colyseus decomposes solutions to these problems into three corresponding components: an *Object Locator*, a *Replica Manager*, and a *Object Placer*. Figure 4 shows the interaction between each component, the game, and components on other nodes. The primary shared interface is the local object store. The game application creates and deletes primaries in the store and periodically executes each primary's think function.

To locate relevant remote objects, we use a robust, distributed *object location* component based on a publish-subscribe system called Mercury (described in Section 4). Mercury allows nodes to register interests using range queries over multiple object *attributes*. Our implementation shows that this is sufficient for Quake II and we believe this is rich enough to support many game genres.

Once discovered, replicas are maintained by a *replica manager* (described in Section 5) which synchronizes replicas with the primary using mechanisms identical to those used by a server to synchronize state with clients (i.e., keep client views up-to-date) in the client-server architecture.

Finally, the *object placer* is responsible for determining where to place primary objects in the system and where to migrate them as a game progresses. We explored a number of migration strategies in Colyseus. Unfortunately, we have found the resultant object migration too disruptive for fast-paced games. Accordingly, we only discuss the object locator and replica manager design in the remainder of the paper. We plan to evaluate the detailed design of object migration in future work. The following sections describe the object locator and replica manager components of

Colyseus in detail.

4 Locating Distributed Objects

Colyseus utilizes a scalable distributed multi-attribute, range-query lookup service for discovering the objects within an area-of-interest. Our implementation is based on the Mercury system design described in Bharambe et. al [12]. Note Colyseus could be adapted to use any underlying system that supports range-query lookup. In this section, we provide a brief overview of Mercury emphasizing aspects relevant to the design of Colyseus along with our extensions to the Mercury design.

4.1 Mercury Overview

The basic service that Mercury provides is a distributed lookup of data items using range-queries over multiple attributes. We call the data items *publications* and queries *subscriptions*. A publication consists of a list of typed attribute-value pairs, similar to a record in a relational database. A subscription is a range query over the attribute-space. Mercury supports distributed range lookups over multiple attributes by partitioning the nodes in the system into groups called *attribute hubs*, one for each attribute in the overall application schema. Nodes within an attribute hub are organized into a circular overlay with each node responsible for a *contiguous range* of attribute values. Subscriptions and publications are routed along hubs for attributes they reference. Much like Distributed Hash Tables (e.g., Chord [27]), Mercury provides a tunable logarithmic bound on the number of hops messages must traverse. The node at which a publication "meets" a matching subscription is called the *rendezvous point* (RP) for that publication. The RP delivers matched publications to the interested subscribers.

Several characteristics make Mercury well suited for object discovery in interactive games. First, multi-attribute, range-based queries efficiently describe object interests such as visible regions of space. Second, object publications and subscriptions naturally exhibit a high degree of locality and predictability (e.g., attributes such as object location exhibit both spatial and temporal locality since objects move in a continuous fashion). Since Mercury stores data *contiguously* in the routing overlay, rather than *randomly* as in a DHT,[6] this locality maps directly onto the overlay, often allowing publishers and subscribers to circumvent the routing paths and deliver their messages directly to the RP by caching recent routes.

4.2 Location and Interest Maintenance

Each game object needs to discover objects within its area-of-interest in order to run its think functions. To locate these objects, each object registers a subscription for objects within its area-of-interest. For example, in Quake II, the subscription describes the region of space that is visible to the primary object from its current location. Objects periodically send events or *publications* containing the current values of its naming attributes, which in this case is its `x,y & z` coordinates.

[6]Mercury maintains load balance across different regions in the overlay by reorganizing range responsibilities and is able to divide responsibilities as well as DHTs. See [12] for details.

Soft State Storage: In the original Mercury design [12], subscriptions are registered as hard-state while publications are transient and not stored at the RP. Colyseus stores both publications and subscriptions as soft state at the RP. The RP expires them after a TTL that each item carries. Unlike Mercury, when a subscription arrives, it matches with all currently stored publications.

This design achieves two goals: First, due to the latency sensitivity of real-time game-play, applications want to discover remote objects as soon as possible. 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 RP. By storing publications, a subscription can immediately be matched to the *recent* publications. This suffices for informing the node about relevant objects due to spatial locality of object updates. Second, some objects in games, such as items, do not change their naming attributes often and it would be wasteful for them to publish their location very often. In general, different types of objects change their naming attributes at different frequencies. Hence the TTL for a soft-state publication is a *tunable* per-object parameter enabling an explicit trade-off between publication overhead and publication staleness.

Attribute Prediction: Another limitation of the design described in [12] is that all publications are represented as points in the attribute space, whereas subscriptions encompass regions. Colyseus generalizes this so that both publications and subscriptions can span ranges. A subscription is matched with a publication if their ranges overlap.

This generalization enables two important features: (1) efficient publication of objects that are large and may not be sufficiently described by a point (e.g., a building); (2) location *prediction* of highly dynamic objects. Rapidly moving objects in the game (for example, players) change their location every frame (10-20 times per second) and hence would need to publish attribute updates very frequently for other nodes to discover them quickly, incurring a substantial publication cost. However, since object movement is largely predictable due to spatial and temporal locality, objects can send *predictive publications* describing their possible future position as a range around the current location. If the prediction turns out to be incorrect (i.e., the object moves outside the predicted range), it only has to publish again. Making the prediction more conservative (e.g., enlarging the range published) reduces this probability but introduces more false matches to subscriptions which overlap extraneous subranges of the publication. Thus, the size of predictive publications is another tunable knob enabled by object predictability, allowing a trade-off between publication rate and false matches.

Interest Prediction and Aggregation: Although Mercury provides $\mathcal{O}(\log n)$ hop routing guarantees, this is insufficient for the interactive response required by games like Quake II. This delay is mitigated to a large extent by the fact that a node only uses Mercury for *discovering* primary objects and not actually for synchronizing replicas it already knows about (See Section 5), so only the replication of objects at the periphery of subscriptions are likely to be delayed and most game applications can probably accept this limited view inconsistency.

However, the same locality that makes predicting location attributes possible also 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 once). Subscription prediction amounts to *speculative prefetching* of object location attributes. Although this speculation may result in extraneous delivery of matched publications, it need not result in unnecessary replication. Upon reception of a prefetched publication, an object 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 subscription locally) before replicating the object. Interest prediction, therefore, enables a tunable trade-off between possible view inconsistency and publication pre-fetching overhead.

Finally, when a node hosts multiple objects, their subscriptions may overlap, especially since many are likely to be spatially nearby (e.g., a player and the missiles it shot). To 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. Colyseus leverages efficient multi-dimensional box grouping techniques originally used in spatial databases [21] to perform this task.

5 Replica Management

The replica management component manages replica synchronization and responds to requests to replicate primaries on other nodes.

Replica Creation and Deletion: When a new object is found that matches a node's interests, the object location component passes the meta-data of the new object (its `GUID`, `EID`, and naming attributes) to the replica manager. The object manager contacts the primary object's owner (identified by its `EID`) directly to register interest in the object (identified by its `GUID`). The primary object's node then marshalls the state required for creating a replica. The primary object's node also periodically sends *deltas-encoded* updates of the object to each replica to keep them synchronized. Each node holding a replica may also send *updates* to the object back to the primary to modify the object. These messages are represented as field-wise diffs encoded with a bit-mask and are marshalled and unmarshalled using the `PackUpdate()` and `UnpackUpdate()` functions shown in Figure 3.

Replicas should be deleted when they are no longer in the area-of-interest of any primary object on the node, since these replicas are no longer of use. Since the replica receives a location update when the object moves, the replica manager can determine if the replica is no longer required by querying each of its primaries' `IsInterested()` functions. Similarly, replicas should be deleted when the corresponding primaries have been deleted. The replica manager identifies these deletions since explicit deletion messages are sent when primaries are deleted. Replicas are maintained as soft-state and are deleted if no updates arrive. This ensure proper behavior even if deletion messages are lost.

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 (<100ms) and game clients can extrapolate or interpolate object changes to present players with a smooth view of the game. 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 tolerate these minor inconsistencies in object store state and 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.

In addition, since the application maintains control over how concurrent local and remote updates are serialized, techniques such as *bucket synchronization* [18], where a small delay is incurred before applying any update to allow for reordering using timestamps, can be used to ensure updates are applied in order with high probability. In addition, nodes can use existing game client interpolation and extrapolation techniques (like dead reckoning [25]) to predict replica state. In our Quake II implementation, we use a simple last-writer-wins protocol. Since state updates distinguish which fields of the object are modified, concurrent updates to the same field are rare, and when they do occur it is usually to fields like object location or velocity that can tolerate small inconsistencies.

Proactive Replication: As discussed in Section 4, Colyseus hides the object discovery latency by pre-fetching objects with predictive publications and subscriptions. However, this solution does not work for short-lived objects such as missiles in Quake II. Since these objects can be created unpredictably and only exist for several seconds, they cannot be efficiently pre-fetched.

However, most short-lived objects in Quake II originate at locations very 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 later will automatically replicate the former, circumventing the discovery phase altogether. For many short-lived objects, like missiles, these annotations are simple to provide since they always originate with predictable naming attributes.

Transparent Pointer Resolution: Objects that contain memory pointers can complicate replication since these pointer values are only valid on the primary's owner. This only becomes a concern if the pointer on a replica is dereferenced by a think function. We found that only a small number of objects in Quake II had this property (e.g., a missile has a pointer to the player that shot it). To avoid having to drastically alter application design practices, Colyseus resolves these references before presenting replica objects to the application.[7] To facilitate pointer resolution, when a replica is marshalled, instead of including a memory pointer, Colyseus marshalls the target object's `GUID` along its owner's `EID`. This allows the receiver to contact the owner to resolve the pointer if required. Recursive pointer resolution of long pointer chains would dramatically increase replication latency, but in practice we found that rarely more than one-level of pointers is required. Moreover, in most cases, an interest will encompass both the object and its pointers (e.g., a missile is almost always spatially close to its creator), so both will be replicated simultaneously. Colyseus maintains *forwarding pointers* [24] to correct stale `GUID` to `EID` mappings (which can occur if objects migrate).

6 Evaluation

In this section, we evaluate the performance of Colyseus using our distributed Quake II implementation. We first evaluate how per-node bandwidth usage scales as the size of the system increases in order to demonstrate that distributed deployments are indeed feasible for large-scale games. This is especially important in a peer-to-peer scenario where bandwidth can be a critical resource. Second, we evaluate the inconsistency penalty incurred by distributing state in Colyseus in order to determine how well our component designs can minimize the interactive "lag" resulting from this

[7] Another option would be to fetch pointers *lazily* only when they are dereferenced by the application. We may add this feature in the future.

inconsistency. Before presenting the results, we describe our modifications to the publicly available Quake II FPS engine [8], followed by the experimental setup and workload used.

6.1 Implementation

In our Quake II implementation, we represent an object's area-of-interest with a variable-sized bounding box encompassing the object's actual visible area.

Our additions to Quake II comprise approximately 3,000 lines of code (excluding serialization code, which we generated automatically). The original Quake II codebase contains approximately 190,000 lines. The majority of the additions deal with interfacing Quake II's data structures with Colyseus' adapter interfaces and managing bounding boxes. We have since incorporated these into a separate utility library useful to any game in three-dimensional space. Nearly all our additions were made at the top-level of the main event loop without changing any of the internal game logic.

Because we automatically delta-encode and serialize Quake II objects using field-wise diffs, 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. For fairness, we also use the unoptimized delta sizes for reporting bandwidth usage in the client-server or broadcast architectures. We note that despite the unoptimized nature of Colyseus' implementation real clients can connect to our distributed servers and play the game with an interactive lag similar to that obtained with a centralized server.

Our current implementation of Mercury provides logarithmic hop routing and supports route caching. However, it does not perform automated load balancing. Each node is currently assigned an equally sized range from each hub. Hence, nodes handling more popular portions of each hub will be responsible for matching a larger number of publications and subscriptions. Nonetheless, our results show that this imbalance is not substantial with this workload.

6.2 Experimental Setup

Emulated Environment: We evaluated Colyseus under 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 across the Internet.

For both scenarios, we emulated the network environment on the Emulab network testbed [28]. The environment did not constrain link capacity but emulated end-to-end latencies by approximately delaying packets using pairwise latencies sampled from the King P2P dataset collected at MIT [6]. The median round trip latency in each experiment was approximately 80-90ms. In the peer-to-peer experiments, we ran 3 virtual servers for each physical Emulab node, while in the federated-server experiments, we ran a single instance of a server per node.

Workloads: In our experiments, game players are simulated with computer controlled bots[8] and get respawned (re-generated) after being killed. Although, we tried several different rule sets, they did not produce qualitative or quantitative differences in our results. Each experiment ran for approximately 15 minutes and, unless otherwise specified, our results examine 5 minutes during the middle of each game. We ran experiments under both the federated and peer-to-peer deployment scenarios.

[8]We used the AceBot bot, written by a third party.

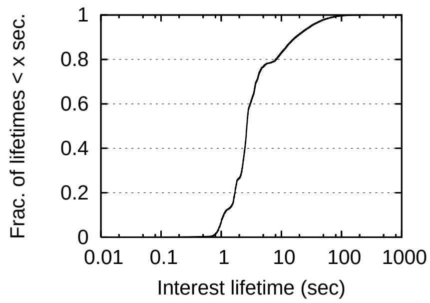
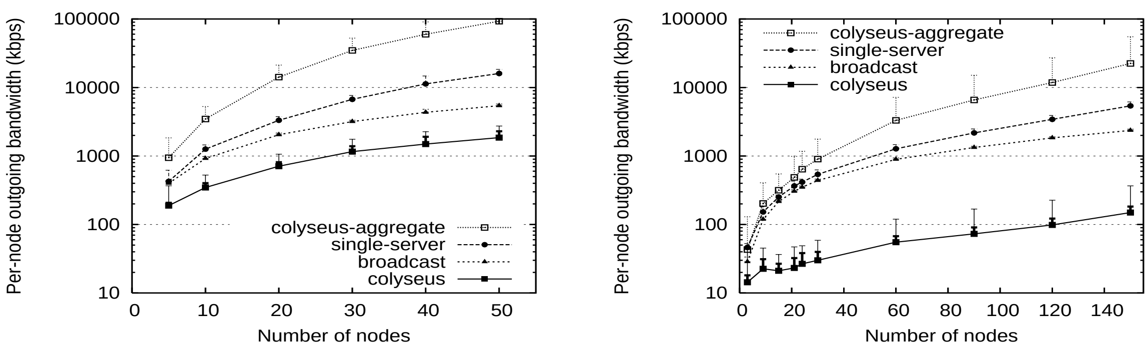


Figure 5: Cumulative distribution of interest lifetimes.



(a) Federated-servers scenario

(b) Peer-to-peer scenario

Figure 6: Scaling in per-node bandwidth usage as the number of servers in the system increase in (a) the federated-servers scenario and (b) the peer-to-peer scenario. (Note the logarithmic scale.)

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

6.3 Colyseus Communication Costs

In this section, we evaluate the communication cost of Colyseus in relation to client-server and broadcast architecture alternatives and examine the characteristics of Colyseus' bandwidth usage as the system scales. Our results for Colyseus are measured value. The results for the client-server and broadcast architectures are based on post-facto simulation using the game traces obtained during the playout of the corresponding Colyseus trial. This ensures that comparison points were based on the same game-play events. This also makes the comparative client-server and broadcast results somewhat idealized. A real implementation may have exhausted system resources and not supported the target frame-rate for game-play.

Scaling Comparison: Figure 6 shows per-node bandwidth scaling as we increase the number of nodes in Colyseus. Each additional node adds 8 players in the federated case (a) and 1 player in the peer-to-peer case (b). Each point on the graph represents the average (across nodes) of the mean bandwidth used by each node in the system. Since game traffic is fairly bursty (see Figure 8), we also plot the 95th percentile of 1 second burst rates as a thin error bar over each point. The thick error bars indicate 1 standard deviation from the mean, which is an relative indicator of load balance. Despite our implementation's lack of dynamic load balancing, the variation is well within an order of magnitude.

For comparison, we plot the bandwidth used by the single server of a client-server architecture and the average bandwidth used by all nodes in a broadcast architecture. We do not include bandwidth consumed by communication to clients in the Colyseus or broadcast cases since we assume that the servers are co-located on the same machine (in the peer-to-peer case) or close to the clients they are serving (in the federated case). If we had included client-server costs, then each server in the federated servers case would only take on an additional 200-220kbps and 10-30kbps in the peer-to-peer case.

In the federated servers scenario (Figure 6(a)), Colyseus' bandwidth usage is a factor of 3-4 less than the broadcast architecture and each node sends a factor of 10 less than the single centralized server. Nonetheless, the aggregate bandwidth used by all Colyseus nodes combined is almost a factor of 10 times more than the total cost incurred by a single server node. In the federated scenario, there are 8 objects per server (8 players) that introduce subscriptions. Each subscription encompasses about 1-2% of the entire game world in our workload. Hence, since objects are placed randomly, each node is always interested in 8-16% or more of the other servers at any given time. Thus, there is significant overhead from object replication in this scenario. However, note that individual servers in this scenario are likely to be better provisioned that typical clients. In addition, since they are near their clients they act as caches for game state with low-latency access (e.g., much like a CDN node), albeit for a non-trivial bandwidth cost.

In the peer-to-peer scenario (Figure 6(b)), Colyseus performs much better relative to the alternative architectures, achieving over an order-of-magnitude improvement over either. Since each node only hosts a single subscribing object (the 1 player), each node is required to know less of the game state than in the federated case. Nonetheless, there is still overhead, as Colyseus uses 5-6 times more bandwidth in aggregate than the client-server scenario. What is more interesting perhaps is that even with 150 players, we are able to support the game with less than 110kbps on average and less than 300kbps in almost all time instances. Keeping in mind that our messages are unoptimized and delta sizes are likely to be smaller for an optimized implementation, this suggests that medium sized peer-to-peer deployment scenarios can be practical with today's broadband technologies.

Communication Cost Breakdown: Note that since we used the same map in the above experiments, the density of the map increases linearly as we increase the number of players. Hence, the per-node communication cost (when dominated by game update traffic) also increases at least linearly,[9] as shown in Figure 6. Figure 7 shows how Colyseus scales with additional servers when we keep the object density constant by spreading 400 players uniformly across all servers. 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. However, we see a 3-fold decrease in communication cost per node with a 5-fold increase in the number of nodes, so the overhead is less than a factor of 2. Nonetheless, it is clear that the returns are diminishing.

Figure 7 also breaks down the communication cost into the different components of Colyseus:

[9]mirroring the quadratic increase in aggregate bandwidth shown in Figure 1(a)

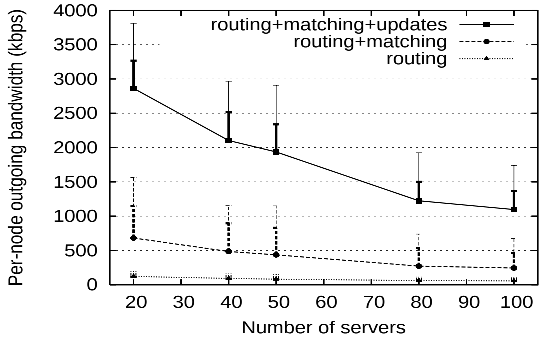


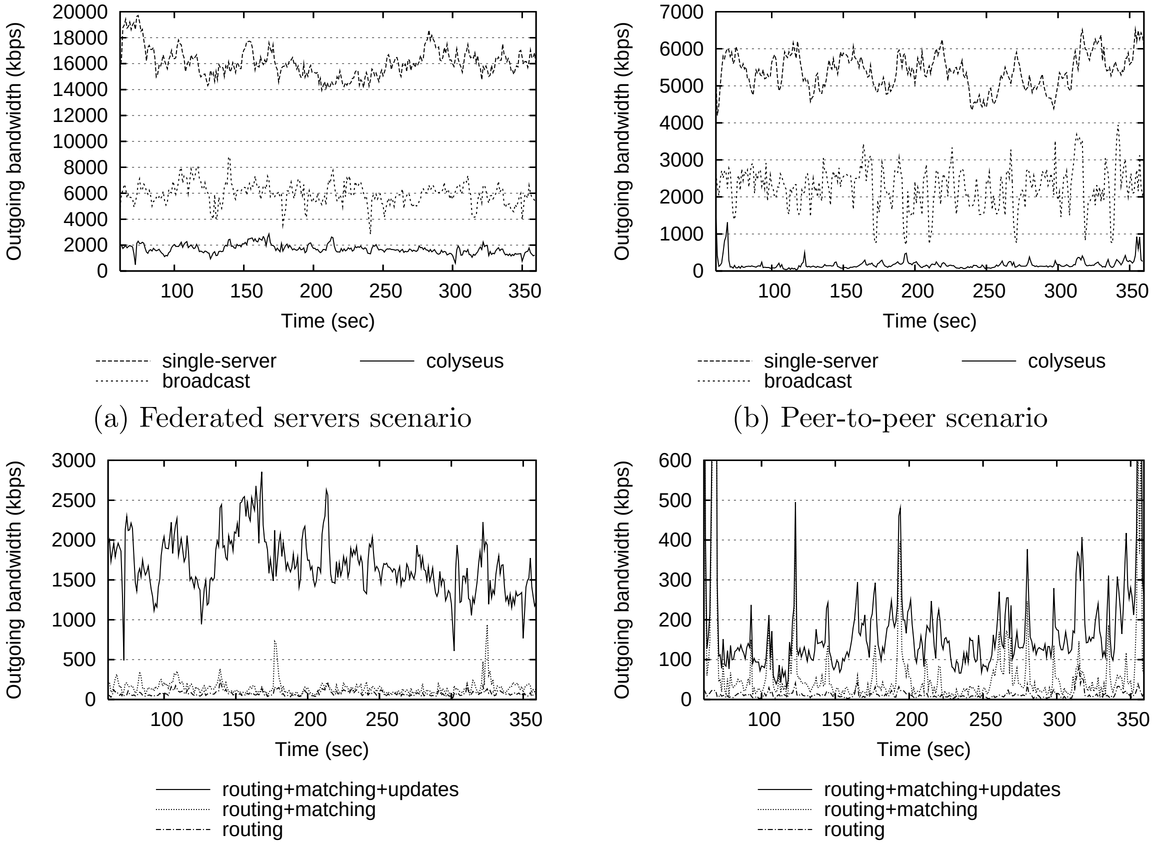
Figure 7: Scaling in per-node bandwidth consumption as the number of servers in the system increases when the number of players is fixed at 400.

sending and routing publications and subscriptions in Mercury (routing), delivering matched publications and subscriptions (matching), and game update traffic (updates). The primary reduction in per-node communication costs, as we add nodes, comes from partitioning game update and Mercury matching traffic. This is expected since we are explicitly dividing the players among more nodes so each node is responsible for sending fewer updates. As we increase the number of nodes participating in the Mercury ring, each node is responsible for matching publications for a smaller portion of the range. Routing traffic per-node also decreases slightly since the linear decrease in publications sent per-node more than offsets the logarithmic increase in forwarding costs. Thus, the addition of additional servers in a federated community deployment can reduce per-node costs.

Traffic Characteristics: To further examine the traffic characteristics of of a game on Colyseus, we examine how bandwidth usage varies over time. Figure 8 (a) and (b) show the outbound bandwidth used by a typical node[10] in Colyseus in the federated and peer-to-peer scenarios, respectively. For comparison, we also plot the traffic characteristics at the server of the client-server architecture and the same node in the broadcast architecture. In all architectures, the traffic is significantly variable across time. This reflects the variable nature of game-play and the fact that game traffic is coupled with the number of visible objects seen by each player when update traffic dominates. In client-server architectures, this variability is usually dealt with by artificially capping the number of objects that can be sent to each client at a time based on an application defined priority ordering. The same solution could be used to limit update traffic in the other two architectures.

Figure 8 (c) and (d) zoom into the Colyseus bandwidth usage time sequence and break down the cost between update traffic and Mercury routing and matching traffic (for the federated and peer-to-peer scenarios, respectively). In the federated scenario, the game update traffic dominates the traffic profile, despite the fact that there are still some bursts in the Mercury traffic. However, in the peer-to-peer scenario, since there is only one player per node, bursts in Mercury traffic dominate the traffic profile. The majority of these bursts occur when many publications are matched with many subscriptions at the node; i.e., when many objects enter the region of the game world corresponding to the node's range in the Mercury ring. We believe that these bursts can be mitigated by distributing the cost of delivering matched publications among the receivers (e.g., by using overlay multicast) at the expense of a little latency.

[10]Here we examined the node with the median overall bandwidth usage.



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

(c) Federated servers scenario (Colyseus) (d) Peer-to-peer scenario (Colyseus)

Figure 8: Figures (a) and (b) compare the outbound bandwidth usage across time between the median node in Colyseus, the same node in a broadcast architecture, and the server in a single-server architecture (in the 50-node federated and 150-node peer-to-peer scenarios, respectively). Figures (c) and (d) zoom into the Colyseus line and shows how much of the total cost is accounted for by Mercury routing and matching (again for the 50-node federated and 150-node peer-to-peer scenarios, respectively). Each point is an average across a 1 second interval. (Note the difference in scale in Figures (c) and (d).)

6.4 Interactive Latency

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. This affects the player's reactions and degrades interactive game-play. 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 9 shows the median time elapsed between subscription generation and new replica creation as we scale the number of nodes in the peer-to-peer scenario (results for the federated servers scenario are similar and are omitted here). The other lines break down the latency into the components required to: (1) route the subscription and a matching publication to the rendezvous point in Mercury (routing), (2) send the match back to the subscriber (matching), and (3) fetch and construct the new replica from the node owning the primary (fetching). The remainder of the discovery delay comes from processing delays and is dependent on the machines in our particular testbed.

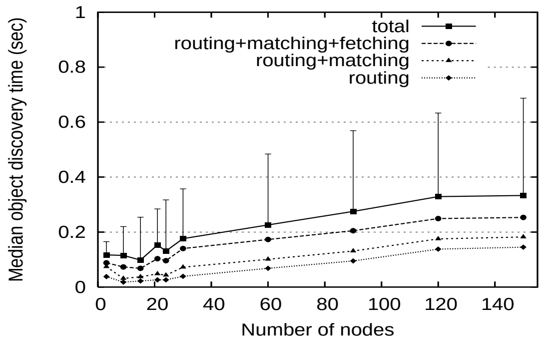


Figure 9: The median time required to discover and construct a replica of an object once a subscription is generated by the application (in the peer-to-peer scenario). The error bars on the top line show the 95th percentile of discovery times.

As expected, the time to deliver a matching publication remains constant at about 1/2 RTTs (40ms) and the time to fetch a replica is about 1 RTT (80ms). The component that scales with the number of nodes is the Mercury routing component which has been shown to be logarithmic in the number of nodes in the system [12]. In absolute terms, with 150 nodes in the system, the median discovery latency is only about 260ms without computational processing overhead and about 330ms with it.

Although interactive latencies exceeding 50-100ms may be noticeable by players, 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, so replica staleness will be tied to the latency distribution of the topology; in our measured Internet topologies, this one-way delay is less than 100ms over 95% of the time. We believe this distinction between discovery and update latency contributes to our observation that the game-play does not feel degraded. We also note that several proximity routing techniques proposed to reduce routing latencies in DHTs [20, 15] could easily be applied to Mercury as well, reducing the latency of the routing component of the discovery time.

Replica View Inconsistency: To examine the impact that object discovery latency has on object store consistency, we examine how often replicas that a node *should* have (i.e., because they are within the bounding box(es) subscribed to by that node) are not present in the node's object store.[11] We compute the fraction of replicas missing at a given time instance as the 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 10 shows the fraction of replicas missing as we scale the number of nodes in the peer-to-peer scenario. For example, with 60 nodes, on average about 6% of the objects we require are missing at any given time and with 150 nodes, a little

[11]Since items do not move, they are much more likely to be globally consistent than mobile objects, so we only examine mobile objects like players and missiles in our evaluation. This will tend to increase the fraction of missing replicas we observe in our results.

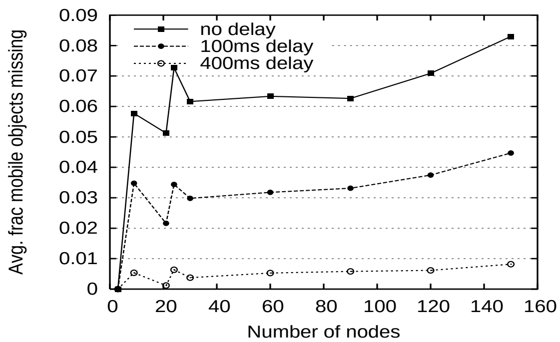


Figure 10: The fraction of replicas missing averaged across all time instances as we scale the number of servers in the peer-to-peer scenario.

more than 8% are missing. The middle (bottom) line shows the fraction missing if we allow 100ms (400ms) to elapse after the precise time we required them. (In Quake II, this corresponds to 1 (4) frames.) Nearly 1/2 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.

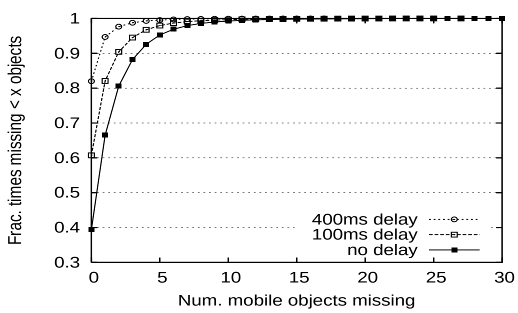


Figure 11: The cumulative distribution of the absolute number of missing remote replicas in 40-node federated servers scenario. Only time instances where at least 1 replica is required are included.

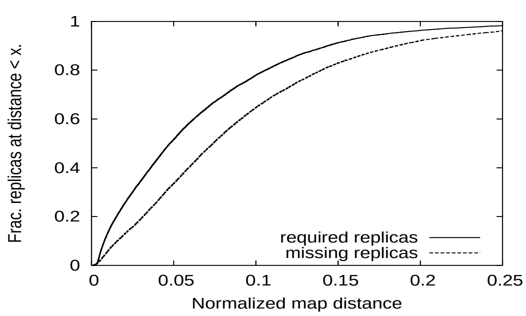


Figure 12: The cumulative distribution of distances from missing and required replicas to interested objects.

Figure 11 shows the cumulative distribution of the number of missing objects for a 40-node (320-players) federated servers scenario. On average, a node required 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 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. We note that for sparser game playouts inconsistency is reduced further. For example, in the 150 node peer-to-peer scenario (150 players), no objects were missing 84% of the time and 98% of the time after 400ms. Moreover, replicas that are missing from a view will tend to be closer to the periphery of object subscriptions (and hence, father away from the interested object and less important). Figure 12 compares the distance of replicas to the objects which expressed interest in them to the distance of those that are missing, across all time instances in the game. Overall, replicas that are missing tend to be farther away. Although the difference in the distributions is not substantial, note that subscription sizes are variable, so objects at the periphery of a subscription may still be close to a player if they are in a small room. More conservative prediction may alleviate this.

Although different games will have different latency and consistency requirements, we believe that our results are promising, especially for games that can tolerate object store inconsistency on the order of a few 100 milliseconds. When inconsistency at the periphery of required regions of space is tolerable, Colyseus may be even more usable.

7 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 including MiMaze [18], Halo [5], and most Real Time Strategy (RTS) games [1], have adopted a *parallel simulation* architecture, where each player in the game simulates the entire game world. Thus, all objects within this world are replicated everywhere and kept consistent using lock-step synchronization. The obvious disadvantages of this architecture are its requirement of broadcasting 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 and the latency between the players. 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. The basic idea is to describe an object's area-of-interest by the cell in which it currently resides and/or a subset of cells nearby. SimMUD [13] is similar to these systems but implements area-of-interest filtering by assigning regions to keys in a Distributed Hash Table allowing it to operate in a distributed environment. Zou, et al. [29] theoretically compared cell-centric approaches with entity-centric approaches, like Colyseus.

The cell-based approach requires that the region-granularity must be specified in advance; in a distributed environment this would either ensure frequent migration of objects between cells if they are small, or force particular nodes to take on a large number of objects if they host large, popular regions. Colyseus differs in that it allows areas-of-interest to be specified in terms of arbitrary bounding boxes. In addition, it does not restrict the placement of objects in the system. These factors allow for efficient communication as well as provide more flexibility for balancing load.

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 able to support FPS games which have much tighter latency constraints as compared to RPGs (which were targeted by SimMUD, for example.)

Several architectures proposed for Distributed Virtual Reality (VR) environments and distributed simulation (notably, DIVE[17], MASSIVE [19], and High Level Architecture (HLA) [22]) 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 of them have been demonstrated to scale to large numbers of participants in practice; DIVE and HLA, for example, originally assumed wide-scale deployment of IP-Multicast.

8 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. Our adaptation of a commercial game (Quake II) to use Colyseus showed that it provides an effective interface for distributing existing game designs. Our evaluation of the Quake II implementation showed that Colyseus is able to: 1) effectively use over a hundred server nodes, 2) support hundreds of game participants and 3) support real-time game-play.

Nonetheless, we should note that the current Colyseus design does not address a few 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. In some deployment scenarios, a distributed architecture also raises issues about cheating. 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.

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] Bharambe, A., Agrawal, M., and Seshan, S. Mercury: Supporting scalable multi-attribute range queries. In *ACM SIGCOMM 2004*.

[13] Bjorn Knutsson and Honghui Lu and Wei Xu and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. In *Proceedings of the IEEE INFOCOM 2004*.

[14] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-area cooperative storage with CFS. In *Proceedings of the 18th Symposium on Operating System Principles* (Oct. 2001).

[15] Dabek, F., Li, J., Sit, E., Robertson, J., Kaashoek, M. F., and Morris, R. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)* (March 2004).

[16] Druschel, P., and Rowstron, A. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th Symposium on Operating System Principles* (Oct. 2001).

[17] Frécon, E., and Stenius, M. DIVE: A scaleable network architecture for distributed virtual environments. *Distributed Systems Engineering Journal 5*, 3 (1998), 91–100.

[18] Gautier, L., and Diot, C. MiMaze, A Multiuser Game on the Internet. Tech. Rep. RR-3248, INRIA, France, Sept. 1997.

[19] Greenhalgh, C., and Benford, S. Massive: a distributed virtual reality system incorporating spatial trading. In *ICDCS '95: Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)* (1995), IEEE Computer Society, p. 27.

[20] Gummadi, K. P., et. al. The Impact of DHT Routing Geometry on Resilience and Proximity. In *Proceedings of the ACM SIGCOMM '03* (Aug. 2003).

[21] Guttman, A. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data* (1984), ACM Press, pp. 47–57.

[22] Institute of Electrical and Electronics Engineers. High Level Architecture. `http://www.dmso.mil/public/transition/hla/index.html`.

[23] Institute of Electrical and Electronics Engineers. Standard for Information Technology, Protocols for Distributed Interactive Simulation. Tech. rep., Mar. 1993.

[24] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained mobility in the emerald system. *ACM Trans. Comput. Syst. 6*, 1 (1988), 109–133.

[25] Pantel, L., and Wolf, L. C. On the suitability of dead reckoning schemes for games. In *NETGAMES '02: Proceedings of the 1st workshop on Network and system support for games* (2002), ACM Press, pp. 79–84.

[26] Shaikh, A., Sahu, S., Rosu, M., Shea, M., and Saha, D. Implementation of a Service Platform for Online Games. In *Proc. of NetGames 2004 Workshop* (Aug. 2004).

[27] Stoica, I., Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the SIGCOMM '01 Symposium on Communications Architectures and Protocols* (2001).

[28] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation* (Boston, MA, Dec. 2002), USENIX Association, pp. 255–270.

[29] Zou, L., Ammar, M. H., and Diot, C. An evaluation of grouping techniques for state dissemination in networked multi-user games. In *MASCOTS '01: Proceedings of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'01)* (2001), IEEE Computer Society, p. 33.