Scalable and Secure Architectures for Online Multiplayer Games

Thesis Proposal

Ashwin R. Bharambe

April 18, 2006

Abstract

Networked games have rapidly evoved from small 4-8 person, one-time play games to large-scale persistent games involving thousands of participants. However, most of these games have centralized client-server architectures which create significant robustness and scalability bottlenecks. This thesis proposes a new modular distributed architecture for interactive multiplayer games. This architecture, called Colyseus, consists of three *independent*, interacting components: Object Placement, Object Discovery and Replica Synchronization. This modularization enables a variety of optimizations which are used critically for load-balancing, for supporting a wide range of consistency requirements and adhering to the tight latency constraints of fast-paced games. This allows to prevent computational and bandwidth hot-spots in the system. Finally, in order to make the system deployable in a peer-to-peer setting, I propose security mechanisms to detect cheating and punish cheating players. I expect these techniques to be applicable to a wide range of multi-party collaborative applications as well.

1 The Problem

Networked games have rapidly evolved from small 4-8 person, one-time play games to large-scale games involving thousands of participants [20] and persistent game worlds. However, they still retain traditional centralized architectures. While such architectures are good for administrative control, security and simplicity, increasingly complex game-play and AI computation is turning the server into a computational and bandwidth bottleneck. Some of the very large games of today (e.g., World of Warcraft [9]) have managed to scale with a centralized server-cluster system, but they have done so at considerable expense and by deliberately slowing down the pace of the game. A client-server game design forces players to rely on infrastructures provided by big game manufacturers, creating a significant barrier-to-entry for small game publishers. Not only that, these infrastructures are sometimes not adequately provisioned (despite the large costs) or long-lived. Thus, they can result in poor performance and/or prevent users from playing their game long after their purchase.

A distributed design can utilize third-party federated servers as well as peer-to-peer deployments to address the above shortcomings. There is considerable evidence [7] that given appropriate incentives, participants are willing to provide resources for such collaborative applications. *The goal of this thesis is the design, implementation and evaluation of a scalable, secure and high-performance distributed architecture for online multiplayer games.*

1.1 Challenges

Architecting distributed applications, in general, has always been difficult. Networked games make this task even harder due to harsher workloads and demands for good performance. The following are the important challenges which any distributed architecture for a networked game must solve:

- In order to relieve the computational bottleneck of a centralized game server, code execution and associated state (required for execution) must be partitioned amongst participating nodes. This partitioning should be efficient, automatic and as *load-balanced* as possible. In addition, subsets of state need to be replicated to ensure timely execution of distributed code. Keeping replicated copies synchronized adds extra bandwidth overhead. This overhead should be minimal and should be load-balanced as well.

- Second, it is important to provide a reasonably consistent view of relevant application state to each node as much as possible. Inconsistency in state translates to a wrongly rendered view for the player and hence directly affects his or her immediate actions. However, since games also involve tight and intense interaction (e.g., rapid movement, fights, etc.), they typically exhibit higher write-traffic and write-sharing than most distributed applications. Guaranteeing reasonable consistency with such workloads is more challenging.
- Third, games suffer from cheating. A distributed architecture lacks a single point of trust which can maintain an authoritative copy of the game state. Hence, mechanisms need to be built-in for policing operations done by players so that game state is kept consistent and correct according to game rules.
- Finally, games demand high performance. Even response latencies of more than 100-150ms are considered bad for the fastest of games (notably, first-person shooters) [2].

The key challenge, therefore, is to re-concile the mutually conflicting objectives of providing high performance and bounded consistency for a write-intensive workload. In addition, a middleware for distributing games should be flexible as well. Games differ widely in terms of their latency tolerance and consistency requirements. Different games (and even different operations within the same game) have differing demands for consistency. For example, players tolerate an occasional missed missile shot in shooting games, however a money transaction in role-playing games (even though it is virtual!) necessarily needs atomicity.

1.2 Drawbacks of Previous Distributed Designs

A number of research [23, 26, 32] and non-research [27, 47] efforts have attempted to address a subset of the above challenges. These approaches have employed one or more of the following techniques.

- *Cell-based Partitioning.* In this approach, the game-world is *a priori* partitioned into a set of (typically, fixed-size) regions called *cells.* Each node is assigned one or more regions, and hosts objects (players and AI entities) present in those regions. This provides a simple way of partitioning state and discovering relevant objects. However, it has two main drawbacks: (a) Regions in a typical game-world exhibit large skews in popularity [7]; furthermore, the popularity distribution changes with time. Thus, a region-based a priori partitioning can result in significant load imbalance. (b) Objects must be migrated when they move between regions. For fast-paced games, this results in high migration rates [7] and connection hand-offs that are likely to be disruptive to game-play.
- *Parallel Simulation.* In order to avoid state partitioning, some games replicate the entire state at each participating node. AI entities are simulated *in parallel* at all nodes, while player objects are synchronized with the master copy (controlled by the player node.) This approach has several drawbacks: (a) Full replication implies inter-node traffic scales quadratically drastically reducing scalability. (b) In addition, parallel executions combined with lost and delayed messages lead to inconsistent states at each node (reported in many popular games [18]) Note that, employing conflict resolution protocols is infeasible given the soft real-time demands of most games.
- *Area-of-Interest Management.* Most virtual reality games are designed such that an individual player interacts with only a small portion (called, *area-of-interest* or `AOI`) of the entire game-world. Thus, only objects within the `AOI` need to be kept up-to-date at all times. Most previous approaches have performed `AOI` filtering by defining the `AOI` to be a *cell* of the partitioned world. Objects subscribe to a multicast channel corresponding to their current cell. Due to limited IP multicast deployment, a form of application layer multicast is used: the node in charge of a cell (*coordinator*) serves as the root of the shared multicast tree for the channel. This implementation suffers from two drawbacks: (a) In addition to computational load imbalance caused by cell-based partitioning (mentioned above), the cell coordinator can also become a bandwidth bottleneck if the cell becomes very popular. (b) Object updates from `nodeA` $\rightarrow$ `nodeB` must flow through the cell coordinator thereby increasing delay. This extra delay can be disastrous for fast-paced games.

These efforts have largely targeted slower games where players tolerate high delays. As such, the focus has been restricted to optimizing computational and communication overheads. Latency performance has mostly been ignored.

1.3 Proposed Solution: Colyseus

This thesis proposes Colyseus, a novel distributed architecture and middleware for online interactive multiplayer games. The following are its two core contributions: (a) *Scalability for Fast-Paced Games.* Colyseus adheres to the tight latency constraints ($\sim$ 100-150ms) demanded by fast-paced games while maintaining scalable communication costs. (b) *Flexible, Modular Architecture.* The Colyseus architecture consists of three independent, interacting components: Object Placement, Object Discovery and Replica Synchronization. This modularization enables a variety of optimizations used critically for load-balancing, for supporting a wide range of consistency requirements and for addressing cheating concerns.

The fundamental reason behind the drawbacks faced by previous approaches is that object placement policies completely dictate how objects are discovered and are kept up-to-date. While this simplifies object discovery, it creates computational and communication load imbalance as well as adds delay to the critical replica synchronization path. Colyseus decouples the object placement and object discovery components. Doing so adds additional overhead for discovering objects. However, it provides several key benefits: firstly, it enables game-developers to tailor object placement to balance computational load and/or minimize inter-node communication. For example, objects interacting frequently might be placed on the same node. Secondly, it permits the use of various object discovery algorithms, e.g., centralized, Distributed Hash Tables (DHT) based, etc. Colyseus, in particular, utilizes a dynamically load-balanced range-queriable DHT called Mercury [4]. This provides a rich distributed query interface and effective pre-fetching subsystem to help locate and replicate objects before they are accessed at a node. Thirdly, it separates the primary-replica synchronization path (which is the critical path determining player-perceived “lag”) from the object discovery path which can potentially be slower. Finally, although Colyseus proposes a single-writer, multiple-reader design for performance, it leaves room for implementing tunable consistency protocols on top, using TACT-like algorithms [45], for example.

Colyseus’ current design addresses scalability and performance. However, for Colyseus to be practical, it should also address the problem of cheating. This is especially important since a completely peer-to-peer scenario is one of our target deployments. The problem can be formulated as Byzantine fault-tolerance for a replicated state machine. This is a very widely studied research topic. However, even the most efficient and practical protocols for achieving Byzantine fault-tolerance [13] are very expensive in terms of bandwidth and latency. Thus, applying them to prevent cheating is quite infeasible. In general, given the need for performance (at least in terms of latency), it seems difficult to prevent cheating altogether. *Hence, this thesis will propose mechanisms to quickly* detect *cheating and blame the involved parties.* The game developer and other parts of the infrastructure can take appropriate actions based on this detection.

Our basic idea is to design a distributed auditing system consisting of the following components: (1) a witness-based *irrepudiable* logging system into which players log their actions and (2) an on-demand auditing system to verify player logs against game rules. Game rules can be obtained using annotations from the game-developer. Another possibility is to re-simulate game actions (also logged) starting from known valid states in the player log and verify that the rest of the log consists of reachable valid states. An important challenge in this part of the work is to define an efficient yet enforceable consistency model. Without a rigorously defined consistency model, it is difficult to differentiate between distributed event orderings which are permissible and which constitute cheating.

A system design is best validated with a real usable system. In order to demonstrate the practicality of Colyseus, I will implement Colyseus and integrate it with the popular Quake II and Quake III first person shooter games. I will evaluate Colyseus using the Emulab [44] network testbed, as well as using a live deployment, if possible.

The rest of the document details the completed and proposed parts of this work plan. The next section provides a short background on game-design and defines some terms. Section 3 describes the Mercury range-query DHT used for object discovery within Colyseus. Section 4 presents the design and an Emulab-based evaluation of Colyseus. Section 5 proposes a high-level design for addressing the cheating problem. Related work is discussed in Section 6. Lastly, we present a work plan, a timeline and summarize the expected contributions of this thesis.

2 Background

In this section, we briefly survey the requirements of online multiplayer games and demonstrate the fundamental limitations of existing client-server implementations. In addition, we provide evidence that resources exist for distributed deployments of multiplayer games. This motivates the exploration of distributed architectures for such games. More details can be found in [7].

2.1 Contemporary Game Design

In a typical multiplayer game, each *player* (game participant) controls one or more *avatars* (player's representative in the game) in a *game world* (a two or three dimensional space where the game is played). This description applies to many popular genres, including first person shooters (FPSs) (such as *Quake*, *Counter Strike*), role playing games (RPGs) (such as *Everquest*, *World of Warcraft*), and others. In addition, this model is similar to that of military simulators and virtual collaborative environments [26, 28].

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 structured as a collection of objects, e.g., 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. 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. Note that this is different from the frame-rate at the client which can be substantially higher to provide a smoother graphics experience. Client-side interpolation is used to create additional intermediate frames.

2.2 Client-Server Scaling Properties

Bandwidth consumed at a game server 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 the game update traffic), `n` ranges from 8 to 64, `s` is $\sim$200 bytes, and `f` is 10 updates per second. A naïve server implementation which simply broadcasts the updates of all objects to all game clients (`c`) would incur an outbound bandwidth cost of $\mathtt{c} \times \mathtt{n} \times \mathtt{s} \times \mathtt{f}$, or 1-66Mbps in games with 8 to 64 players.

Commercial servers do better, of course. Two common optimizations are employed: *area-of-interest* filtering and *delta-encoding*. Since, individual players typically only interact with a small portion of the game world at any one time, only updates about *relevant* objects are sent to the clients. 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. These optimizations reduce `n` and `s` respectively. In the case of 8-64 player Quake II games, server bandwidth consumption reduces to about 62-492 kbps.

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

As the computational load on a server increases, the server may require more than 1 frame-time of computation to service all clients. Hence, it may not be able to sustain the target frame-rate. Figure 1(a) shows the mean number of frames per second *actually* computed by the server, while Figure 1(b) shows the bandwidth consumed at the server for sending updates to clients. We note several points: First, as the number of players increases, area-of-interest filtering

[1]Some large scale RPGs use multiple servers. They partition the world into disjoint regions so that players on different servers cannot interact. Hence, they remain similar to the client-server architecture and are still centralized.

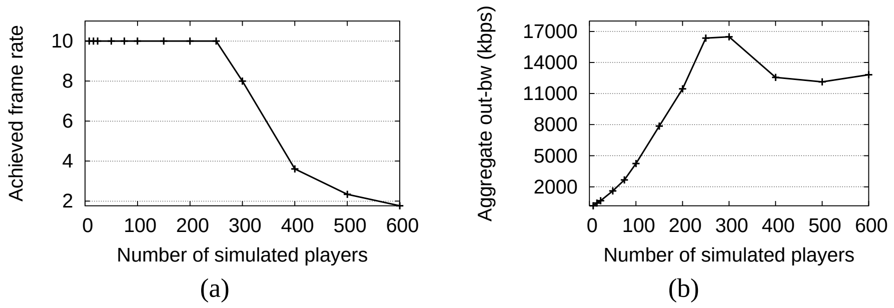


Figure 1: Computational and bandwidth scaling at the server end of client-server system.

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 Rationale for Distributed Deployments

The scaling limits shown in Figure 1 are clearly dependent on the capability of the server employed. Thus, these limits can be raised by utilizing more powerful servers or server-clusters. Many research designs [41], middle-ware layers [8, 27, 46] and a few commercial games [36, 39], in fact, use this approach. This approach affords many advantages – simplicity and tight administrative control being the most important. However, it requires large investments for hardware, bandwidth and maintenance creating a significant barrier-to-entry for small game publishers or grass-roots gaming efforts. A widely distributed game implementation, on the other hand, can (1) address the scaling challenges, (2) provide more fault-tolerance and (3) can make use of existing third party federated server deployments that we describe below.

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

Table 1: Availability of third-party federated server deployments.

There is significant evidence that given appropriate incentives, players are willing to provide resources for multiplayer games. For example, most FPS game-servers are run by third parties (such as “clan” organizations formed by players). Table 1 shows the number of active third-party servers we observed for several different games. Older games (e.g., Quake II) typically have much fewer servers than recent ones (e.g., CounterStrike). We also found (see [7] for details) that often, more than 50% of the servers were idle, having no active players. The server count and utilization suggest that there are significant resources that a distributed game design may use.

With this motivation, in this thesis, 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. Such a widely distributed gaming architecture must address unique problems, such as inter-node communication costs and latencies. A solution to these problems forms the core of the proposed thesis.

3 Distributed Range Queries with Mercury

This section presents the design and evaluation of the Mercury distributed location service which Colyseus uses to discover distributed game objects. Mercury provides a dynamically load-balanced routing protocol for querying objects using *multi-attribute range queries*. Much recent work on building scalable peer-to-peer (P2P) networks has concentrated on Distributed Hash Tables or DHTs [37, 40, 43]. DHTs offer a number of scalability advantages over previous P2P systems (e.g., Napster, Gnutella, etc.) including load balancing and logarithmic hop routing with small local state. However, the hash table or "exact match" interface offered by DHTs, although fruitfully used by some systems [11, 14, 17], is not flexible enough for many applications. In our model, each query is a *conjunction of ranges* in one or more attributes. The attributes not present in the query are assumed to be wildcards. We believe that range query support offered by Mercury significantly enhances search flexibility: in addition to being useful for answering user queries, it can also be useful in the construction of distributed applications as shown by Colyseus. Colyseus' use of Mercury is described in more detail in Section 4.

3.1 Completed Work

We now present an overview of the Mercury design, and a brief simulation-based evaluation. More details can be found in an associated paper [4]. I have implemented Mercury as part of the Colyseus implementation, and have publicly released it [5].

3.1.1 Data Model

In Mercury, data items are represented as a list of typed attribute-value pairs, very similar to a record in a relational database. Each field is a tuple of the form: $(\texttt{type}, \texttt{attribute}, \texttt{value})$. The following types are recognized: $\texttt{int}, \texttt{char}, \texttt{float}$ and $\texttt{string}$. A query is a *conjunction* of predicates which are tuples of the form: $(\texttt{type}, \texttt{attribute}, \texttt{operator}, \texttt{value})$. A disjunction is implemented by multiple distinct queries. Mercury supports the following operators: $<, >, \leq, \geq$ and $=$. For the `string` type, Mercury also permits *prefix and postfix* operators. This query language could be used to specify regions of interest in a real or virtual space, for example. Figure 2(a) presents an example.

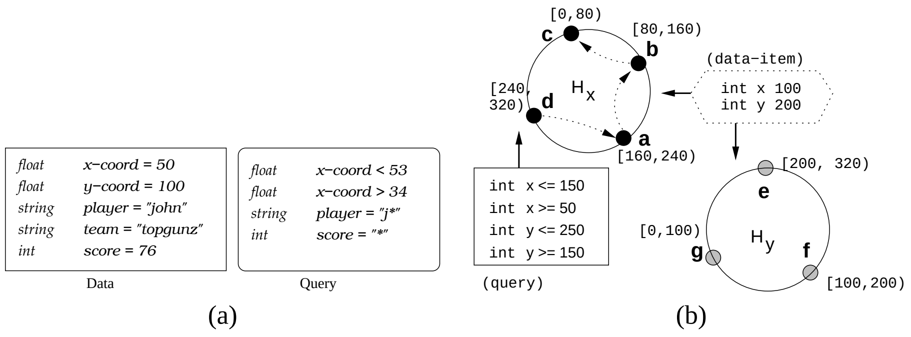


Figure 2: (a) Example of a data-item and a query. (b) Routing of data-items and queries in Mercury.

As described above, data items are persistent, while queries "stream through" the network. Instead if queries are persistent, they are termed *subscriptions* and data items are called *publications*. Thus, Mercury can be thought of as a distributed *publish-subscribe* system as well.

3.1.2 Routing Overview

Mercury supports queries over multiple attributes by partitioning the nodes in the system into groups called *attribute hubs*. Each attribute hub is responsible for a specific attribute in the overall schema. For routing queries and data-items, the first routing hop determines which hub to route through. The rest of the routing happens in a single hub and is based on the value of a single attribute. Nodes within a hub are arranged into a circular overlay with each node responsible for a *contiguous range* of attribute values. In addition to having links to nodes within its hub (similar to other DHTs), each node must also maintain a link to each of the other hubs. We expect the number of hubs for a particular system to remain low, and, therefore, do not expect this to be a significant burden.

Queries are passed to exactly one of the hubs corresponding to the attributes that are queried. Although choosing any attribute hub suffices for matching correctness, substantial savings in network bandwidth can be achieved if the choice is done more intelligently using query selectivity. Within the chosen hub, the query is delivered and processed at all nodes that could potentially have matching values. To guarantee that queries locate all the relevant data-records, a data-record D, when inserted, is sent to all hubs corresponding to the attributes present in D. Within each hub, the data-record is routed to the node responsible for the record's value for the hub's attribute. For routing queries, we route to the *first* value appearing in the range and then use the contiguity of range values to spread the query along the circle, as needed. Figure 2(b) shows an example.

3.1.3 Constructing Efficient Routes

Like Symphony [34], the key to Mercury's route optimization is the selection of k *long-distance* links that are maintained in addition to the successor and predecessor links. As a result, each node has a routing table of size $k + 2$ including its neighbors along the circle. k is a configurable parameter here and could be different for different nodes. The routing algorithm is simple: Let neighbor n_i be in-charge of the range $[l_i, r_i)$, and let d denote the *clockwise distance* or *value-distance* between two nodes. When a node is asked to route a value v, it chooses the neighbor n_i which minimizes $d(l_i, v)$. Let m_a and M_a be the minimum and maximum values for attribute a, respectively.

Then,

$$d(a,b) = \begin{cases} b - a & \text{if } a \le b, \\ (M_a - m_a) + (b - a) & \text{if } a > b \end{cases}$$

A node n whose value range is $[l, r)$ constructs its long-distance links in the following fashion: Let I denote the unit interval $[0, 1]$. For each link, a node draws a number $x \in I$ using the *harmonic* probability distribution function: $p_n(x) = 1/(n \log x)$ if $x \in [\frac{1}{n}, 1]$. It contacts a node n' (using the routing protocol itself) which manages the value $r + (M_a - m_a)x$ (wrapped around) in its hub. Finally, it attempts to make n' its neighbor. As a practical consideration, we set a fan-in limit of $2k$ links per node. We will refer to a network constructed according to the above algorithm as a ValueLink network.

Define *node-link distance* between two nodes a and b on the circular overlay as the length of the path from $a \rightarrow b$ in the *clockwise* direction. Under the assumption that node ranges are uniform, node-link distance is directly proportional to value-distance. In this case, we can prove (see [34]) that the expected number of routing hops for routing to any value within a hub is $\mathcal{O}(\frac{1}{k} \log^2 n)$. This guarantee is based upon Kleinberg's analysis of small-world networks [31]. Unfortunately, the "uniform node ranges" assumption can be easily violated for many reasons. Firstly, ranges are assigned in a distributed manner without using cryptographic hashes which means they are not guaranteed to be uniform. More importantly, explicit load-balancing would cause nodes to cluster closely in parts of the ring which are popular. Mercury utilizes a distributed histogram maintenance scheme based on light-weight random sampling to provide efficient routing even with highly non-uniform ranges.

In order to create links using the correct harmonic distribution, Mercury utilizes node-count histograms. These histograms estimate the number of nodes present in various regions of the circular overlay. Given such a histogram, long distance links are formed as follows: first, the number of nodes n in the system is estimated. For each long-distance link, a value n_l between $[1, n]$ is generated using the harmonic distribution. This represents the number of nodes that must be skipped along the circle to reach the desired neighbor, `N`. The histogram is then used to estimate a value v_l that `N` will be responsible for. Finally, a join message is sent to this *value* v_l which will get routed to `N` using the existing overlay.

3.1.4 Novel Algorithms in Mercury

Random Sampling Mercury's approach for sampling utilizes the fact that the hub overlay is an *expander graph* [35] with high probability. Hence, random walks on this network converge quickly to the stationary (uniform) distribution. Thus, a node can use the end-point of a short random walk (with a TTL of $\mathcal{O}(\log n)$, for example) as a representative random node in the system. It is important to note here that efficient random sampling in a distributed network is non-trivial [33], especially when routing is *not* based on node identifiers. In the case of Mercury, no special overlay is needed for performing random sampling (as opposed to Ransub [33]); in fact, sampling messages can be easily piggy-backed on keep-alive traffic between overlay neighbors.

Approximate Histograms The above sampling algorithm is used to construct approximate histograms for various system-wide metrics. The basic idea is to estimate the required distribution *locally* and exchange these estimates throughout the system in an epidemic-style protocol. Each node periodically samples k_1 nodes uniformly at random using the random-walk based algorithm described. These nodes report back their local estimates along with the most recent k_2 estimates they have received. As time progresses, every node builds a list of tuples of the form: $\{\mathtt{node\_id}, \mathtt{node\_range}, \mathtt{time}, \mathtt{estimate}\}$. Each of these tuples represent a point on the required distribution – stitching them together in a metric-specific manner yields a piecewise linear approximation.

Histograms thus made are utilized within Mercury for many purposes. As discussed in the last subsection, they are used in the construction of the overlay itself. They are also used for faster load-balancing and for estimating query selectivity.

Query Selectivity Recall that a query is sent to only one of the attribute hubs corresponding to the attributes queried. Also, a query represents a *conjunction* of its predicates each of which can have varying degree of selectivity. For example, some predicate might be a *wildcard* for its attribute while another might be an exact match. A wildcard predicate will get flooded to every node within its attribute hub. Thus, the query should be sent to that hub for which it is *most selective* to minimize the number of nodes that must be contacted.

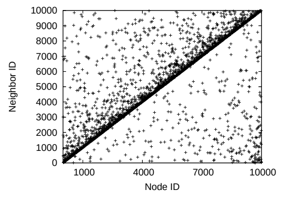
The problem of estimating the selectivity of a query has been very widely studied in the database community. The established canonical solution is to maintain approximate histograms of the number of database records per bucket. In our case, we want to know the number of nodes in a particular bucket. Each node within a hub gathers such an histogram for its own hub using the histogram maintenance mechanism described above. In addition, using its inter-hub links, it can also collect histograms for other hubs. These histograms are then used to determine the selectivity of a subscription for each hub.

Dynamic Load Balancing In many applications, a particular range of values for an attribute may exhibit a much greater popularity in terms of database insertions or queries than other ranges. To avoid popular nodes from getting swamped, Mercury uses a leave-rejoin load balancing algorithm which uses load histograms. First, each node can measure if how its local load compares with the average load in the system using these histograms. Second, the histograms contain information about which parts of the overlay are lightly loaded. Using this information, heavily loaded nodes can quickly discover lightly loaded parts of the network. A light node is asked to gracefully leave its location in the routing ring and re-join at the location of the heavy node. A leave-join exchange takes place between two nodes if $\mathtt{Load}_{\mathtt{heavy}} \geq \alpha.\mathtt{Load}_{\mathtt{light}}$. The above protocol *eventually* load balances all nodes i.e., for any node, $\alpha l > \bar{L}$ and $l < \alpha\bar{L}$, for $\alpha > \sqrt[3]{4}$. Here, l denotes the load of a node and $\bar{L}$ denotes the average load in the system. Over time, leaves and re-joins result in a node-range distribution such that range spans become inversely proportional to their popularity.

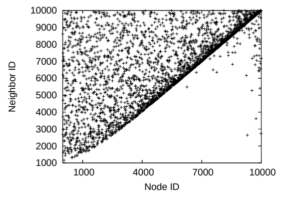
3.1.5 Evaluation

This section presents a simulation-based evaluation of the Mercury protocol, focusing only on routing scalability. Other more detailed results can be found in [4]. In what follows, n will denote the number of nodes within a hub; we concentrate on routing within a single hub only. Every node establishes $k = \log n$ intra-hub long-distance links. We assume without loss of generality that the attribute under consideration is a `float` value with range $[0, 1]$. NodeLink denotes the ideal small-world overlay, i.e., long distance links are constructed using the harmonic distribution on node-link distance. ValueLink denotes the overlay when the harmonic distribution on value-distance is used (Section 3.1.3.) HistoLink denotes the scenario when links are created using node-count histograms. Note that the performance of the ValueLink overlay is representative of the performance of a plain DHT (e.g., Chord, Symphony) under the absence of hashing and in the presence of load balancing algorithms which preserve value contiguity.

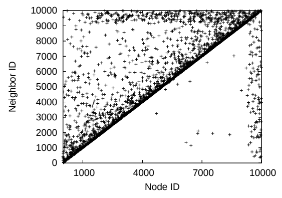
For evaluating the effect of non-uniform node ranges on our protocol, we assign each node a range width which is inversely proportional to its popularity in the load distribution. Such a choice is reasonable since load balancing



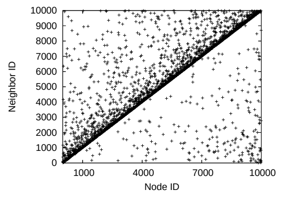
(a) NodeLink



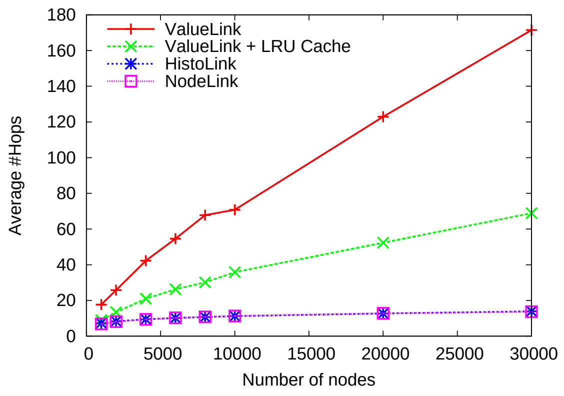
(b) ValueLink



(c) HistoLink [1:1]



(d) HistoLink [$\log n$: $\log n$]



(e)

Figure 3: Plots (a)-(d) provide details about the links constructed by various link construction algorithms. For HistoLink, $[k_1 : k_2]$ means k_1 nodes were queried per round each giving k_2 estimate reports; 5 exchange rounds were performed. Plot (e) compares their scalability in terms of the average number of routing hops.

would produce precisely such a distribution – more nodes would participate in a region where load is high. The ranges are actually assigned using a Zipf distribution. In particular, data values near 0.0 are most popular and hence a large number of nodes share responsibility for this region, each taking care of a very small node range. For reference, in our simulator setup, these are also the nodes with lowest numeric IDs.

Figure 3(e) compares the performance of the protocol with and without approximate histograms to guide the selection of the long-distance links. We see that the NodeLink and HistoLink overlays perform much better than the ValueLink overlay. These effects are explained using Figure 3(a)-(d) which plots the distribution of long-distance links. Recall that, in a ValueLink overlay, nodes construct their links by routing to *values* generated using a harmonic distribution. However, in this case node ranges are not uniformly distributed – in particular, nodes near the value 1.0 (i.e., nodes with higher IDs) are less popular, so they are in charge of larger range values. Hence, the long-distance links they create tend to *skip* over less nodes than appropriate. This causes all the links to crowd towards the least popular end of the circle. Hence, packets destined to these nodes take circuitous routes along the circle rather than taking short cuts provided by the long-distance links. Although caching ameliorates the effect, the performance is still much worse as compared to the optimal NodeLink overlay. On the other hand, we see that the performance of the HistoLink overlay is nearly the same as that of the optimal NodeLink overlay. Again, looking at Figure 3(a)-(d), we find that node-count histograms enable nodes to establish a correct link distribution (corresponding to the NodeLink overlay) quickly using very low overheads.

3.2 Planned Work

The basic Mercury architecture has been extended by Colyseus in many important ways. These extensions will be discussed in the next section. However, since Mercury is a general purpose routing substrate, it's usage is not restricted to Colyseus only. This creates a number of possibilities for extending Mercury. One enhancement is to utilize application layer multicast protocols for publication delivery from the rendezvous point. Another is to incorporate QoS support for publication delivery for supporting heterogeneous client populations. However, I believe these extensions are outside the scope of this thesis. Since the Mercury implementation has already been released publicly, I believe it will allow other researchers to easily add extra features. For example, it has been successfully used for the design of a distributed defragmented DHT-file system called D2 [29] which organizes data-blocks *contiguously* using Mercury instead of spreading them across random machines using consistent hashing. This has been shown to provide significant availability and performance gains for users accessing entire files or set of files.

4 Colyseus Architecture: Design and Evaluation

There are two types of game state, immutable and mutable. We assume that immutable state (e.g., map geometry, game code, and graphics) is globally replicated since it is updated very infrequently, if at all. Colyseus only manages the collection of mutable objects (e.g., players' avatars, computer controlled characters, doors, items), which we call the *global object store*. The Colyseus architecture is an extension of existing game designs described in Section 2.1. In order to adapt them for a distributed setting, Colyseus partitions the global object store and associated think functions amongst participating nodes. It replicates (and pre-fetches) relevant state needed for the timely execution of think functions.

4.1 Completed Work

This section starts with an overview of the Colyseus design, followed by a description of its three components: object locator, object placer and replica synchronizer. Then, we present an evaluation of this architecture using a real game on the Emulab testbed. More details can be obtained from [6, 7].

4.1.1 Overview of Colyseus Design

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

In practice, the node holding the primary can synchronize replicas the same way viewable objects are synchronized on game clients in client-server architectures. If we assume that messages between two nodes are never delivered out-of-order, the above replication model provides per-object sequential consistency, also called cache coherence [16]. Modifications made to a secondary replica are 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.

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

The execution of a think function may require access to objects that a node is not the primary owner of. Although a think function could access any object in the game world, *most think functions do not require access to all of them.*[2] To ensure correct execution, a node must create secondary replicas of required objects. Fetching these replicas on-demand could result in a stall in game execution, violating real-time game-play deadlines. Instead, Colyseus mandates each primary object to predict the set of objects that it expects to read or write in the near future, and prefetches the objects in the read and write sets of all primary objects on each node. The prediction of the read and write set for each primary object is specified as a selective filter on object attributes, which we call an object's *area-of-interest* or `AOI`. We believe that most games can succinctly express object `AOI`s using range predicates over multiple object attributes. Such predicates work especially well for describing spatial regions in the game world. For example, a player's interest in all objects in the visible area around its avatar can be expressed as a range query (e.g., $10 < x < 50 \wedge 30 < y < 100$). In summary, at each node, Colyseus maintains replicas that are within the union of the `AOI`s of the primaries in the local object store.

Object Discovery Colyseus uses both traditional randomized DHT and *range-queriable* DHTs (described earlier in Section 3) as scalable substrates for object location. Range-queries describing `AOI`s, which we call *subscriptions*, are

[2] This is because, objects in a virtual reality game (in order to mirror the physical world) have fundamentally limited "sensing" abilities.

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

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

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

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

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

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

This architecture does not address some game components, such as content distribution (e.g., game patch distribution) and persistent storage (e.g., storing persistent player accounts). The problem of distributing these components is orthogonal to distributing game-play and is readily addressed by other research initiatives [17, 19].

4.1.2 Architecture Components

Colyseus implements the preceding design with three components: an *object locator* (Section 4.1.3), which implements a range-queriable lookup system for locating objects within AOIs, a *replica manager* (Section 4.1.4), which maintains and synchronizes replicas with primaries, and an *object placer* (Section 4.1.5), which decides where to place objects and migrate them during game-play. Although all three components are implemented in Colyseus, our focus will be on the first two components in this thesis. We assume an object is placed on the closest node to the controlling player, which is likely optimal for minimizing interactive latency. Figure 5 shows the interaction between each component, the game application, and components on other nodes.

4.1.3 Object Locator

To locate objects, Colyseus implements a distributed location service on a DHT. Unlike other location services built on DHTs [10, 42], the object locator in Colyseus must be able to locate objects using range queries rather than exact matches and must handle a continuous and changing query stream from each participant in the system. Moreover, data

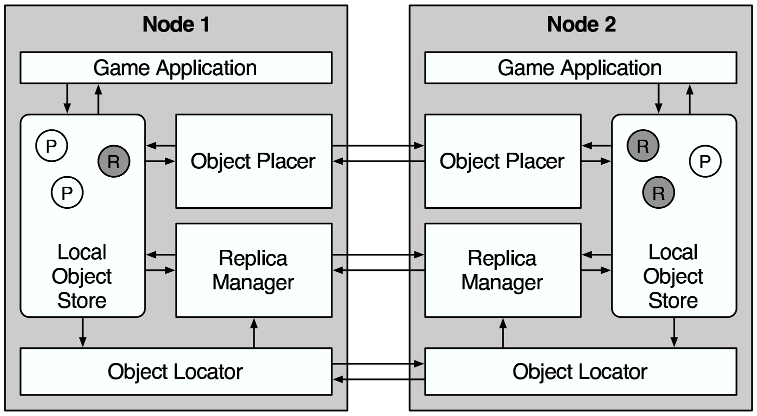


Figure 5: The components of Colyseus. Circled R's represent secondary replicas, circled P's represent primary objects.

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 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.

Traditional vs. Range-queriable DHTs With a traditional DHT, the object locator bucketizes the map into a discrete number of regions and then stores each publication in the DHT under its (random) region key. Similarly, subscriptions are broken up into DHT lookups for each region overlapping with the range query. A *range-queriable* DHT like Mercury (Section 3) may be an even better fit to a distributed game architecture. Unlike traditional DHTs which use discrete random keys (to achieve load balance), a range-queriable DHT stores key-values contiguously on the overlay. This allows range queries to be expressed directly, instead of having to be broken up into multiple DHT lookups. Moreover, object location metadata and queries are likely to exhibit spatial locality, which maps directly onto the overlay, allowing the object locator to circumvent routing paths and deliver messages directly to the rendezvous by caching recent routes. Finally, since nodes balance load dynamically in a range-queriable DHT to match the publication and subscription distribution, they may be able better handle the Zipf-like region popularity distribution observed in Section 2.

Colyseus implements both these object location mechanisms, and we evaluate the trade-offs in [7]. We find that a range-queriable DHT like Mercury achieves better scalability and load balance than a traditional DHT when used as an object location substrate, with a small consistency penalty.

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. We use a simple moving average of an object's velocity for prediction, and special cases are made if more is known about an object's physics (e.g., missiles always move in a straight line). A small factor (PubTime) is added to account for the discovery and delivery time of publications for objects entering the object's subscription volume. Subscription prediction amounts to *speculative pre-fetching* 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 pre-fetched publication, a node can cache (for the length of the TTL) and periodically check whether it *actually* desires the publishing object (by comparing the publication to its up-to-date unpredicted 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 further reduce subscription overhead, Colyseus enables aggregation of overlapping subscriptions using a local *subscription cache*, which recalls subscriptions whose TTLs have not yet expired (and, thus, are still registered in Mercury), and an optional *aggregation filter*, which takes multiple subscriptions and merges them if they contain sufficient overlap. This filter uses efficient multi-dimensional box grouping techniques originally used in spatial databases [25].

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

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

4.1.4 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 object on the node responsible for its region (as in *cell*-based architectures [27, 32, 39]). However, FPS game workloads exhibit rapid player movement between cells, which entails migration between servers. For example, in a 500 player game on 100 servers (see [7]) 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 [32]). However, this approach adds at least one extra hop for each update. The resultant delays are significantly worse than sending updates directly point-to-point, especially considering the target latency of 50-100ms in FPS games [2].

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

Table 2: Percentage of object updates that can bypass object location.

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. Table 2 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 4). Any node interested in the latter will automatically replicate the former, circumventing the discovery phase altogether.

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 3: Impact of proactive replication on missile object inconsistency.

Table 3 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' `AOIs` 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, $<$100ms for 93% of node pairs in the topology we used), and is *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 [45], 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 [2]. 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.

4.1.5 Object Placement

We propose an object placement scheme based on black-box inference of object interests. In this scheme, each node constructs a partial *interest graph*, where objects are vertices and interests between objects are directed edges, which we call *interest links*. Interest links are inferred using the subscription each object provides. Edges are weighted by their *cost*; for example, in our current system, they represent the communication cost associated with maintaining remote interests. Based on this information, the object placement component on each node uses simple heurisitics to independently reduce the (weighted) number of interest links to remote nodes. Due to changes in interest patterns or an increase in load, a node may need to off-load a subset of its objects to another lighly loaded machine. To support the discovery of a lightly loaded node, we use a Mercury hub organized by the load attribute of a node. The general concept is similar to that used in Abacus [30] to improve application performance, though we exploit properties of interests in games to work with arbitrary graphs.

Details The object placement component views object placement as an optimization problem. Here we present one possible cost model based on communcation cost, which is the the limiting factor in a distributed game architecture. However, we believe it is possible to use models that account for other costs as well (such as CPU load). We call the cost that a node expends its *load*. Our goal is for the load, $\mathtt{load}_\mathtt{n}$ on each node, $\mathtt{sid}_\mathtt{n}$, to remain between a low water mark, `lowwater`, and a high water mark, `highwater` (which is less than the node's capacity, to enable it to absorb bursts.)

To facilitate the construction of the interest graph, each node must be able to infer what the "real" interests of each object are. We take a simple approach in our current system and say that an object `X` has a *stable* interested in object `Y` if `X` has maintained an interest in `Y` for a time greater than some threshold. This threshold can be dynamically set based on interest history, or based on known application semantics. Object costs, can either be determined by the application developer or measured directly. We have observed that message sizes are very predictable for objects in Quake II, so we use fixed values for these costs based on our measurements.

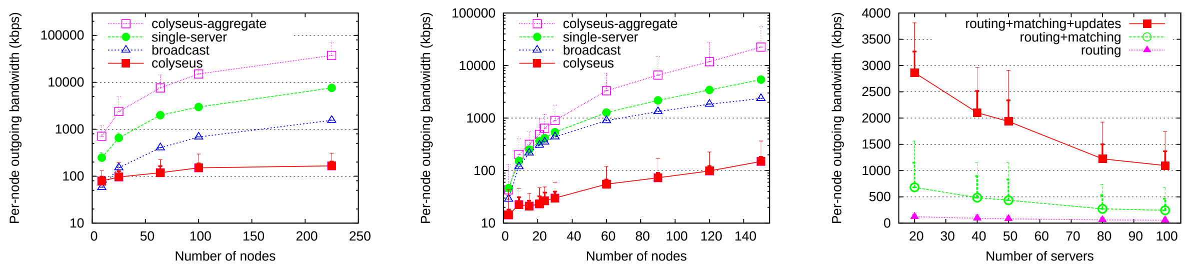
Finally, to achieve the goal of reducing the number of replicas in the system, we essentially want to *cluster* mutually interested objects on the same node, thereby, limiting the number of remote interest links. In addition, we use a separate *load shedding* heuristic to offload objects from a node when it is too heavily loaded (i.e., above `highwater`.) However, this heuristic must also try to avoid increasing the number of replicas in the system to reduce the danger of oscillations resulting with the clustering heuristic.

Discussion Unfortunately, while the mechanisms are intuitive and shown to be useful for certain workloads, with a real Quake II workload, we observe continuous oscillations. Fundamentally, FPS games are very fast-paced and the object interests do not remain stable enough for performing clustering. However, we note that interest inference is a useful general purpose mechanism for games and similar applications; it can be fruitfully used for balancing load when the workload is less dynamic as compared to FPS games. Also, *any* object placement scheme can be "plugged into" Colyseus since the object manager functions properly regardless of the location of objects in the system; object migration is an optimization. More details about the heuristics and our evaluation can be found in [6].

4.1.6 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. 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 for our Quake II evaluation. We use the Mercury `rangedht` as the object location substrate, and linearize the game map when mapping it onto the DHT. Further details about the setup, the Quake II prototype and additional results can be found in report [6, 7].



(a) Model Workload (b) Quake II (c) Quake II (Same Density)

Figure 6: 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.

Communication Cost

Figure 6 compares the bandwidth scaling of Colyseus running p2p games with the client-server and broadcast architecture alternatives. We simulate the alternatives using the same game-play events as the real execution on Colyseus.

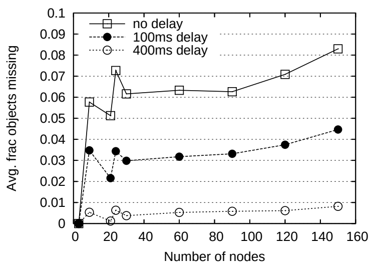
Figure 6(a) shows the scaling properties with rectangular maps under our model workload. This workload was based on game-parameters estimated from a set of Quake III games played by real players. More details about this workload can be found in [7]. The workload keeps mean player density constant by 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.

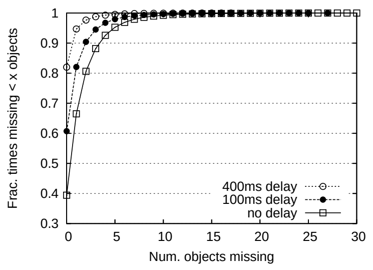
Figure 6(b) shows the same figure when running with the Quake II workload. We observe similar scaling characteristics here, except that the per-node Colyseus bandwidth appears to scale almost quadratically rather than less-than-linearly as in our model workload. This is primarily due to the fact that 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 6(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.

View Inconsistency

We now examine the view inconsistency, i.e., fraction of missing local replicas, observed in the Quake II workload Figure 7(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 7(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.


(a) Consistency Scaling


(b) 40-fed

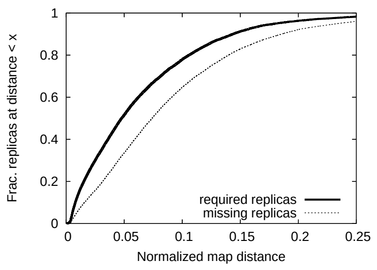

(c) Missing Replica Locations

Figure 7: (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.

4.2 Planned Work

Our evaluation shows that Colyseus is able to meet the latency requirements of FPS games like Quake II, maintain scalable communication costs and keep the view inconsistency to a minimum. A number of questions still remain regarding Colyseus' design and evaluation. We discuss these next.

4.2.1 Measuring View Inconsistency

Although we find that 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 7(c) compares the distance (over time) 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. I plan to explore a more game-play-centric evaluation of view inconsistency. In particular, I plan to use the quality of the video (measured in terms of PSNR) resulting from the game stream as a metric of evaluating inconsistency. This metric is likely to capture the importance of a game object for the final rendered output.

4.2.2 Deployment and Workload Gathering

I have integrated Colyseus with the open-sourced Quake II game, however since the game is old, it is not very popular. I intend to integrate it with Quake III now (this could not be done earlier since the game was open-sourced very recently), and deploy it to a wide audience. Performance measurements obtained from this deployment will be used to validate the design choices made in Colyseus as well as serve as workload for future research.

4.2.3 Applicability to Other Game Genres

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 [15]. 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. If possible, I plan to apply Colyseus design techniques to a real MMORPG game and validate them.

4.2.4 Consistency Models

Colyseus adopts a simple primary-backup replication model which provides per-object sequential consistency. While we have argued uptil now that this model is sufficient for FPS games like Quake, that is not true for other types of games. For example, virtual money transfers in a MMORPG need transactional semantics. Even for FPS games, there are situations where stronger consistency models are necessary. An exploding missile which kills regardless of player health, for example, must either kill everybody in a room or leave everybody unharmed. The explosion action thus needs to be *atomic*. The Colyseus model also implies that updates from two primary nodes may be received at a node in an arbitrary order. Hence, it is possible to witness a player die before the killer missile explodes! In such circumstances, it is desirable to have events delivered in a *causally consistent* order.

As part of this thesis, I intend to explore appropriate consistency models and lighter-weight versions of distributed consistency protocols (e.g., variants of causal serializability [38]) for Colyseus. It is hoped that Colyseus' flexibility in object placement will be used crucially in the design of these protocols. Note that rigorous consistency models are also needed for formalizing the problem of cheating (discussed in the next section.) For example, the consistency model adopted will decide which distributed event orderings are legitimate (truly concurrent), and which are a result of malicious behavior.

5 Security in Colyseus: Ensuring Cheat-proof Playouts

We have seen that Colyseus promises good scalability in terms of per-node computation and bandwidth costs without sacrificing latency performance. These guarantees are unfortunately not enough for networked games. Players in a game make every effort to *cheat* to gain an unfair advantage over other players. If a game permits easy cheating, players quickly lose the incentive to compete. Thus, it is important, even crucial, for a game design like Colyseus to ensure cheating possibilities are rare. However, while decentralization provides us increased scalability, the lack of a central point of trust in the system creates new possibilities for cheating. In this section, we present examples of the types of cheats that the Colyseus architecture makes possible and outline a generic strategy for addressing them. *This section represents a major part of the planned work in this thesis.*

5.1 Cheat Classification

Recall that there are three components in Colyseus: an object locator, an object placer and a replica manager. Security vulnerabilities in each of these components can result in cheats. Some of these are similar to the cheats in other architectures, while others are unique to Colyseus.

Object Locator Related Cheats: If object lookup requests (subscriptions) are not correctly routed, objects needed for rendering a view will be missed and hence game-play will be affected. There is an incentive for being malicious: by hijacking a player's subscriptions, the enemy can become 'invisible' for the player when entering the latter's viewable arena. Similarly, players can send incorrect (larger than their correct `AOI`) subscriptions and receive objects which game rules typically disallow. For example, a player may be able to see objects in farther areas of the game world which may give him or her an unfair advantage.

Object Placer Related Cheats: Players may be able to take advantage of the object placer's decisions. For example, if the primary copy of a nearby missile object is located on the same node as the player, the player can tinker with the execution of the missile's think function in order to gain an advantage (by sending invalid "kill" writes to another player's object, for example.)

Replica Manager Related Cheats: The replica manager design is also susceptible to cheating attacks. Note that each object has a single primary situated on some node, and all writes to the object pass through this node. In particular, for a P2P design, each player is in charge of the primary copy of its own game object! This design gives unprecedented and total control of the object to the primary node: the node can perform arbitrary writes, can ignore valid writes sent from other replicas and can perform an arbitrary serialization of writes. For example, the player node could always claim that it received the `kill` update from a missile later than its own `update_position` update. This would ensure the player is never killed.

In this thesis, we shall focus for the most part on the latter two cheat types. We believe that it is possible to leverage security architectures for DHT systems [12, 21] to perform secure object lookups with high probability. Improper lookups (subscribing to large parts of the world) can be thwarted by intermediate routing nodes if they look at the current position of the subscribing player. Another possibility is to utilize the witness nodes (described in the next subsection) to generate subscriptions.

5.2 Proposed Design

Our focus is to avoid cheats in the object placer and replica manager in Colyseus. This problem can be rephrased as follows: we want to ensure that *writes* to distributed objects are applied in accordance with game rules. Notice that it is possible to formulate this problem as an instance of a fault-tolerant replicated state machine. Rules of the game define valid transitions of this state machine. Each node in the system acts both as a possibly byzantine client and a possibly byzantine server, performing read and write operations on the global game state. Protocols like BFT [13] can be applied to the problem since they provide safety and liveness even in the presence of byzantine clients. Unfortunately, such protocols are quite expensive, especially in terms of the latency constraints they force. As such, using these is not feasible for games, particularly fast-paced ones. In general, given the need for high performance, it seems impossible to *prevent* cheating altogether. Therefore, the emphasis in this thesis will be more on *detecting* cheating in a timely manner as opposed to *preventing* it.

5.2.1 Design Overview

At a high level, our design can be thought of as a *distributed auditor.* With each node `A` in the system, we associate another randomly chosen *witness node* $\mathtt{W_A}$, which oversees the operations of node `A`. All operations performed by node `A` are logged securely at the witness. These logs can then be *audited* at a later stage to detect incorrect behavior. They can also be used to selectively rollback application state, as far as is feasible.

Notice that our design necessitates that a witness does not have any vested interest in the witnessed node. In other words, the witness should neither be malicious nor collude with the witnessed node. During the first phase of this study, we will assume that a randomly picked witness is sufficient for the above purpose. If a witness is chosen such that an attacker cannot maliciously witness a node of its choice, it is possible to make this non-collusion model practicable. Picking a random node can be accomplished in various ways. For example, if we assume a DHT substrate exists for object lookup, witness for a node `A` can be defined as $\mathtt{successor(hash(IP_A))}$. A pseudo-random hash function will ensure that the witness is selected uniformly at random from the network. It also ensures that a malicious witness node cannot choose a particular node to spy on. The above simple choice is not without weaknesses. I plan to explore how much vulnerability this causes. I also plan to explore the challenges of generalizing the model to allow byzantine behavior by the witness.

Witnesses and Logging In order to audit player actions at a later point in time, it is necessary to *log* all updates done by a player. Furthermore, the player should not be able to tamper with the logs after he has modified the game state. In our design, the witness plays the role of ensuring the authenticity of the log. In the current design, we achieve this is by sending all updates via the witness node.

Consider a `player` object whose primary is hosted at node `B`, as shown in Figure 8. All *writes* to `player` (including those done by its primary node `B`) are sent through the witness $\mathtt{W_B}$ for node `B`. Thus, when a `missile` object on node `A` performs a write to the `player` replica on node `A`, this update is sent in encrypted form to witness $\mathtt{W_B}$. The witness orders updates and sends them over to node `B` where they are committed. The witness thus acts as the serialization point for updates to the `player` object. This design adds additional delay to the primary → replica update path since updates done by the primary itself also need to be sent to the witness. To alleviate this, we propose that the primary send "optimistic" writes directly to the replica at the same time it sends them to the witness. Because the witness may end up invalidating some updates, in some cases, the optimistic writes can result in object inconsistency for a small window of time (approximately `1.5 RTT`.) However, we expect such situations to be infrequent.

Writes are encrypted using a shared key established between each replica and the primary node. Encryption ensures that the witness cannot maliciously re-order or drop updates using information within the writes. The witness can still behave maliciously in other ways; Section 5.2.2 discusses this in more detail. The witness also stores the encrypted updates in its log for later auditing. To prevent a malicious witness from lying and claiming it does not have some updates which it really does have, witnesses are required to send a signed acknowledgement back to the witnessed node. That node keeps a log of these acknowledgements, to prove that updates did indeed reach the witness.

Detecting Cheats using Audits Most current games of today are structured as state machines where the actions performed by players and AI entities correspond to transitions within the game state. These transitions can be utilized to check the validity of local game states maintained by nodes (i.e., states of the primary objects in their local object stores) in the system. We briefly discuss the challenges of deriving such rules for real games in Section 5.2.2.

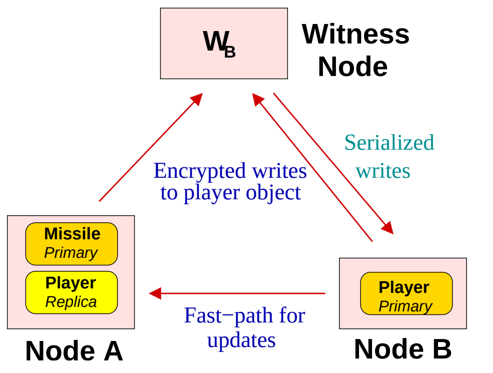


Figure 8: Flow of distributed object writes in the witness model.

Given a set of such rules, players can run a *proof-checker* in the background to check if a player they are interacting with is potentially cheating. No extra communication is required in this case since the player will already have replicas of interacting players which are updated every frame. Such checking is not possible, however, for players which are not interacting with any other player. In such cases, it is the responsibility of the witness node to randomly perform a sanity check for the state reported by the witnessed node. Since the witness already has a log with encrypted updates, it can just ask the node to reveal its encryption key for the updates it sent in the past. The node is then asked to update its private encryption key so that the witness cannot tamper with newer updates.

We assume the existence of a centralized trusted entity in the system, which can perform the role of an auditor. The auditor can perform the above-mentioned cheat-detection upon request. Trust in the auditor node can be used to impose penalties on cheating nodes (by isolating them from the game, for example.) Note that, in our system model, we assume nodes have IDs obtained through an authentication procedure which associates credit cards or other personal information with the ID. This is needed in order to prevent Sybil attacks, and is a natural fit for the business model for online games as well.

5.2.2 Challenges

A number of challenges must be addressed in order to make the above high-level design practical. Firstly, in the witness model, all updates made to a local object must be sent to the witness node for serialization. This is quite expensive. Hence, we need to explore ways to make it more efficient. One possibility is for the node to only send periodic *commitments* of the current state to the witness. A commitment for frames $f_1 \rightarrow f_1 + \Delta$ is a hash over the states of primaries and replicas for each of these frames. Thus, the node can tamper with its log only within a set of Δ frames. Δ allows a tradeoff between bandwidth overhead and cheating flexibility available to the node.

Another important challenge is the fact that nodes may feign delayed receipts of messages. For example, a player may claim it received an update from a closing door only after he had made his move. In general, a node may not apply certain updates coming from other primaries to its local replicas. The root cause of these problems is the lack of game-level synchronization and the weak consistency model provided by Colyseus. By exporting stricter consistency models and appropriate synchronization primitives, it is possible to alleviate such cheats. However, care must be taken to avoid stalls in game execution due to frequent locking.

Finally, automatic cheating detection using transition rule checking is non-trivial, as well. The problem is that some of the transition rules (e.g., motion physics of travelling missiles) can be quite complex. One possibility is to use the game developers' insights for approximating such complex rules. For example, instead of verifying whether a position update was absolutely correct (i.e., precisely following game physics) at every frame, it might suffice to say that a player with certain capabilities cannot move more than x units in a particular dimension in 10 frames. Also, non-determinism in state transitions (for example, when random number generators might be used to decide AI moves), needs to be captured for verifying state transitions.

6 Related Work

In this section, we list the designs adopted by current games, as well as previous research architectures for distributed games. See Section 1 for a detailed discussion on their drawbacks and how the Colyseus design improves upon them.

Some games (e.g., MiMaze [23], Halo [1], and most Real Time Strategy (RTS) games [3]) use a *parallel simulation* architecture, where each player in the game simulates the entire game world. All game objects are replicated everywhere and kept consistent using lock-step synchronization. The obvious disadvantages of this architecture are its broadcasting of updates to every player, resulting in quadratic bandwidth scaling behavior, and its need for synchronization, limiting response time to the speed of the slowest client. These deficiencies are tolerated in RTS games because individual games rarely involve more than 8 players.

Second-Life [39] and Butterfly.net [27] perform interest filtering by partitioning the game world into disjoint regions or cells. Much like Colyseus' DHT-based lookup, SimMUD [32] makes this approach fully distributed by assigning cells to keys in a DHT. Zou, et al. [48] theoretically compare cell-centric approaches with entity-centric approaches, like Colyseus. While our results show that such a design works well, it does have some weaknesses. The granularity of the cells must be chosen to carefully match the typical area-of-interest size. This may be difficult in some games where the area-of-interest size varies widely.

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

Several architectures proposed for Distributed Virtual Reality (VR) environments and distributed simulation (notably, DIVE [22], MASSIVE [24], and High Level Architecture (HLA) [26]) have similar goals as Colyseus but focus on different design aspects. DIVE and MASSIVE focus on sharing audio and video streams between participants while HLA is designed for military simulations. None address the specific needs of modern multiplayer games and, to our knowledge, none have been demonstrated to scale to large numbers of participants. For example, DIVE and HLA originally assumed wide-scale deployment of IP-Multicast.

7 Work Plan and Timeline

This section summarizes the completed and remaining work in this thesis. The items followed by □ are considered necessary for a minimum acceptable thesis; those followed by $\star$ are part of the expected thesis. Bonus items are marked with $\star\star$. Work already done is marked by a ✓, and items that I plan to leave to future researchers by †.

- ► The Mercury range-queriable DHT
  - ▷ Design of the routing protocol supporting scalable range-query routing in distributed systems with dynamic load-balancing. ✓
  - ▷ Simulation-based evaluation of the scalability of the system. ✓
  - ▷ Implementation of the Mercury protocol with all features. Public release for other researchers. ✓
- ► Colyseus Architecture for Multiplayer Games
  - ▷ Design of the basic architecture consisting of three major components: Object Discovery, Object Placement and Replica Management. ✓
  - ▷ Implementation and evaluation on the Emulab testbed using a toy-game (for controlling various parameters) and using a real FPS game (Quake II). ✓

  - ▷ Development of variants of stronger consistency models (causal consistency, atomicity) for applicability to other games. □†
  - ▷ Evaluation using a deployed Quake III implementation. ⋆
  - ▷ Workload characterization using the deployed Quake III implementation. ⋆⋆
  - ▷ Better metrics for measuring view inconsistency in games. ⋆†

- ▶ Security in the Colyseus Architecture
  - ▷ Develop a witness-based system for irrepudiable logging and auditing. □
  - ▷ Implementation within Colyseus and integration with the Quake III implementation. ⋆
  - ▷ Generalization to byzantine witness behavior. ⋆⋆†

Work Item	**Approximate Time frame**
Development of newer consistency and anti-cheat protocols	Apr 06 → Jun 06
Integrating Colyseus with Quake III	Apr 06 → Jun 06
Implementation of anti-cheat protocols within the witness-based scheme	Jun 06 → Aug 06
Implementation of PSNR computation	Jun 06 → Jul 06
Deployment, trace-gathering and evaluation of Quake III	Jul 06 → Oct 06
Possible submission to NSDI 2007	Oct 06
Work on bonus items and writing	Oct 06 → Jan 07

Table 4: Timeline for completion of planned work

Table 4 shows the rough timeline planned for the completion of the work items presented above.

8 Expected Contributions

I expect this thesis to make the following contributions:

- The Mercury range-queriable DHT. This is the first distributed routing protocol providing range-query support *and* dynamic load-balancing. One important contribution of the Mercury system is its use of light-weight sampling techniques to estimate various system-wide metrics for performing many optimizations. Mercury has also been implemented. This artifact is another related contribution of the thesis. This will allow researchers to use it as a building block and make improvements to it. It has already been used for a "defragmented" distributed file system [29] for guaranteeing higher availability and performance.

- Design, implementation and evaluation of the Colyseus architecture for distributed interactive multiplayer games. Colyseus is the first distributed design to be successfully applied for scaling fast-paced FPS games which demand tight latency bounds. At the same time, the architecture is flexible, permits various load-balancing algorithms and permits a variety of consistency models to be built on top. Thus, it can serve as a scalable substrate for a gamut of virtual reality games.

- A large portion of the Colyseus design has been implemented and integrated into a real game (Quake II). I intend to integrate with the more popular Quake III game, deploy it and present measurement results. This evaluation will be one of the first large scale evaluations of networked games. The workload and client resources data generated will be valuable for further research.

- Finally, the anti-cheating measures developed for Colyseus will encourage more deployments of distributed architectures for networked applications. I hope these measures will also lead to a family of lighter-weight Byzantine fault-tolerance protocols providing high performance at the cost of weaker guarantees (sufficient for certain scenarios.)

References

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

[2] Beigbeder, T., et al. The Effects of Loss and Latency on User Performance in Unreal Tournament 2003. In *NetGames*, August 2004.

[3] Paul Bettner and Mark Terrano. 1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond. *Gamasutra*, March 2001.

[4] Ashwin Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: Supporting scalable multi-attribute range queries. In *SIGCOMM*, August 2004.

[5] Ashwin Bharambe and Jeffrey Pang. Mercury Implementation v0.9.2. `http://www.cs.cmu.edu/~ashu/research.html#mercury`.

[6] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. A Distributed Architecture for Interactive Multiplayer Games. Technical Report CMU-CS-05-112, CMU, January 2005.

[7] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: A Distributed Architecture for Online Multiplayer Games. In *Proceedings of the 3rd NSDI*, May 2006.

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

[9] Blizzard Entertainment, Inc. World of Warcraft. `http://www.worldofwarcraft.com`.

[10] Luis Felipe Cabrera, Michael B. Jones, and Marvin Theimer. Herald: Achieving a Global Event Notification Service. In *HotOS 2001*.

[11] M. Castro, P. Druschel, A. M. Kermarrec, A. Nandi, A. Rowstron, and Singh A. Splitstream: High-bandwidth multicast in a cooperative environment. In *Proceedings of the 19th Symposium on Operating System Principles*, October 2003.

[12] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Security for structured peer-to-peer overlay networks. In *5th Symposium on Operating Systems Design and Implementaion (OSDI'02)*, December 2002.

[13] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computing Systems*, 20(4):398–461, 2002.

[14] Castro M., et al. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE J. on Sel. Areas in Comm.*, 20(8), October 2002.

[15] Chen, K., et al. Game Traffic Analysis: An MMORPG Perspective. In *NOSSDAV*, June 2005.

[16] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems. Concepts and Design*. Addison-Wesley, New York, NY, 2001.

[17] Dabek, F. et al. Wide-area cooperative storage with CFS. In *SOSP*, October 2001.

[18] Diablo Cheats. `http://thegw.com/cheats/diablo.html`.

[19] P. Druschel and A. Rowstron. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, October 2001.

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

[21] Amos Fiat, Jared Saia, and Maxwell Young. Making Chord Robust to Byzantine Attacks. In *European Symposium on Algorithms*, 2005.

[22] Emmanuel Frécon and Møarten Stenius. DIVE: A scaleable network architecture for distributed virtual environments. *Dist. Sys. Eng. J.*, 5(3):91–100, 1998.

[23] L. Gautier and C. Diot. MiMaze, A Multiuser Game on the Internet. Technical Report RR-3248, INRIA, France, September 1997.

[24] Greenhalgh, C., et al. Massive: a distributed virtual reality system incorporating spatial trading. In *ICDCS*, June 1995.

[25] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, June 1984.

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

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

[28] IEEE. Standard for Information Technology, Protocols for Distributed Interactive Simulation. Technical report, March 1993.

[29] Jeffrey Pang, et al. Defragmenting DHT-based Distributed File Systems. Private Communication.

[30] Khalil Amiri and David Petrou and Gregory Ganger and Garth Gibson. Dynamic Function Placement for Data-intensive Cluster Computing. In *USENIX Annual Technical Conference*, 2000.

[31] Jon Kleinberg. The Small-World Phenomenon: An Algorithmic Perspective. In *Proceedings of the 32th ACM STOC*, 2000.

[32] Knutsson, B. et al. Peer-to-peer support for massively multiplayer games. In *INFOCOM*, July 2004.

[33] Dejan Kostic, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin Vahdat. Using random subsets to build scalable network services. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, March 2003.

[34] Manku, G. et al. Symphony: Distributed Hashing in a Small World. In *USITS 2003*.

[35] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

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

[37] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network . In *SIGCOMM 2001*.

[38] M. Raynal, G. Thia-Kime, and M. Ahamad. From Serializable to Causal Transactions for Collaborative Applications. Technical Report RR-2802, INRIA, France, February 1996.

[39] Philip Rosedale and Cory Ondrejka. Enabling Player-Created Online Worlds with Grid Computing and Streaming. *Gamasutra*, September 2003.

[40] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale p2p systems. In *Middleware*, November 2001.

[41] Shaikh, A., et al. Implementation of a Service Platform for Online Games. In *NetGames*, August 2004.

[42] Stoica, I., Adkins, D. et al. Internet Indirection Infrastructure. In *SIGCOMM 2003*.

[43] Stoica, I., et al. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, August 2001.

[44] White, B., et al. An integrated experimental environment for distributed systems and networks. In *OSDI*, December 2002.

[45] H. Yu and A. Vahdat. Design and Evaluation of a Conit-based Continuous Consistency Model for Replicated Services. *ACM Trans. on Comp. Sys.*, August 2002.

[46] Zona, Inc. `http://www.zona.net`.

[47] Zona Research. Internet business review: Keynote systems. `http://www.keynote.com/services/assets/applets/zona_keynote.pdf`, 2000.

[48] Zou, L., et al. An evaluation of grouping techniques for state dissemination in networked multi-user games. In *MASCOTS*, August 2001.