Object Placement in Distributed Multiplayer Games

Jeffrey Pang Justin Weisz

December 12, 2003

Abstract

We propose two distributed optimization heurisitics for object placement in a distributed environment, and evaluate the communication cost incurred by several object placement strategies in the context of distributed First Person Shooter multiplayer games. We show that our heurisitics do as well or better than placement strategies that place objects statically or partition objects based on game region, achieving relatively low communication cost, even load distribution, and a lower migration frequency. Our estimates show that such a system could be used on servers with symmetric 1.5Mbps links and would incur a global communication cost of less than 3 times that of using a single server.

1 Introduction

In the past decade there has been an explosion in the popularity of online multiplayer games. However, most of these games are still designed using a client-server model, with only ad hoc techniques for distributing responsibility (for example, splitting the virtual world into "realms" or "zones"). Moreover, there has been little formal analysis or evaluation of the scalability of alternate solutions.

We evaluate methods for scaling a particular genre of games called First Person Shooters (FPS). FPS games are unique in that their game state is highly dynamic and state updates must be propagated in a timely manner. Moreover, due to the high cost these constraints impose on servers, the typical FPS still only supports 16-32 players per game. We choose this genre to demonstrate the utility of our placement strategies even in a tightly constrained game model.

The dominant model for provisioning servers in an FPS is to allow individuals and small organizations (e.g., "clans") to run servers independently. This model has several nice properties: the cost for provisioning is distributed among participants, and these servers are more likely to have good connectivity and sufficient capacity than clients. For example, CounterStrike [6], a game originally developed by players themselves, has an estimated 35,000 servers deployed [7]. However, while the number of servers is usually plentiful, load (e.g. the number of players on each server) is almost always significantly imbalanced because clients participating in the same game must play on the same server. Examining game servers for several FPS games shows that for many games, over 60% of the servers have *no load*, while a non-trivial fraction are near or at capacity.

We would like to harness these free resources by distributing a single game over multiple servers, allowing a fairer allocation of resources and much larger scale games to play out. However, there is currently no model for *object placement* in such a system. This paper evaluates several object placement strategies for a distributed FPS game.

2 System Design

The goal of our system is to provide a framework for distributing game objects across servers. This framework serves as an interface for traditional game server logic but transparently optimizes the placement of objects in the distributed system.

2.1 Game Server Logic

An FPS game server like Quake 2 is implemented as a discrete event loop (each iteration of which is called a *frame*) which processes the actions and interactions of the objects it maintains (such as client commands, physics, etc.); we call this set of objects the *object store*. We assume that the server logic is agnostic to its history of operations and cares only about the current contents of its object store (persistent state is only saved in objects managed by the store) and some immutable static state (e.g., the map); hence the actions executed are defined solely in terms of the current contents of the store.

Thus, we can divide the objects in the store among multiple servers, as long as each server has all the objects required to execute the objects it owns. This can be approximated since an object is only *interested* in some region of the game space (for example, its visible region) and can not be affected by objects outside that region. Nonetheless, there will always be objects which need to be in multiple partitions (such as when a player is in a hallway between two larger groups of players). In this case, the object can be assigned to one of the servers, while other just maintains a copy (see Section 2.3 for details).

Our framework intelligently manages the object store, optimizing the placement of objects in the system with these constraints in mind. Because a server cares only about the objects in its store in a given frame, we can do this optimization with only one additional piece of information: a *subscription* describing the portion of the game space in which each object is interested. This will typically fall out automatically from game semantics (e.g., a bounding box in 3D space, or, in the case of Quake 2, the set of BSP tree nodes that are potentially visible).

2.2 Communication Architecture

To support dissemination of information between objects on different nodes, we utilize a distributed *publish-subscribe* system, such as Mercury [4]. In such a system, nodes specify a set of subscriptions to indicate their interests. Nodes disseminate updates via publications, which are matched with subscriptions and routed to the

interested parties. Our heuristics will function on any such system, but for our evaluation, we assume a *rendezvous-point* based system like Mercury because of its scalability properties. In such a system, publications and subscriptions are routed to an independent rendezvous-point where they are matched.

2.3 Object Model

Dynamic game objects (players, missiles, items, etc.) have a *primary* copy located on a single node which is responsible to serializing updates for that object. Remote updates (by objects residing on other servers) are sent directly to the node on which the primary is residing. For flexibility, ordering and application of updates, both at primaries and replicas, is done in an application specific manner (see Section 2.6). Though remote updates are possible, most updates to an object in a game are made by "itself" (e.g. movement).

Clients: When an object represents a player (e.g., the entity that player controls), a client-connection is associated with that object. A clients only send commands (e.g. the equivalent of keystrokes) to control the object and does not modify its state directly.

Replicas: A node obtains a replica of an object when it subscribes to a space containing it. The primary occasionally sends out updates to its state as publications to synchronize replica state. The state required for representing a replica is small, and it is often possible to send *deltas* of the state rather than complete snapshots (e.g., in current games, replica state is sent from the server to clients; delta sizes in Quake 2, for example, are on average 16 bytes per 100ms for player objects). Hence, it is possible to send publications relatively often (on the order of 10 or 20 times a second). This is especially important in FPS games where low latency observation is critical for playability. For scalability in a distributed environment, replicas are maintained as soft-state; when a node does not receive an update for a replica for a fixed period of time, it is discarded.

Object Location: A node must obtain the location of a primary to be able to update it. Since only objects that are interested in a primary can update it, we piggy back the current location of an object in publications, so servers will obtain the a pointer to the primary location when they obtain a replica through subscription.

2.4 Object Structure

Each node maintains a set of primary `Objects`, a set of replica `Objects`, and a list of other servers it knows about (along with load statistics). Because our object placement heuristics depend on inferred *interests* between objects, we also maintain `InterestLinks` between objects which form an *interest graph* (i.e., relations between objects which indicate that one is interested in receiving information about the other).

Servers keep the following information about each `Object`:

type: indicates primary or replica status.
properties: application specific fields (health, ammo, etc.).
subscriptions: the space(s) of interest.
node: where the primary resides.
interests: objects we are interested in.

Each `Interest` in **interests** comprises:

object: the object that the interest describes.
start_time: the time we became interested in this object.
end_time: the time we stopped being interested in this object.

```
Object {
    ModifySubscription(new_sub);
    value GetProperty(key);
    ModifyProperty(key, value, type);
    virtual ReceiveUpdate(update_delta) = 0;
}
```

Figure 1: Game object interface.

We obtain interests by examining replicas on our node (we can determine if a replica is in a primary's subscribed space). This can be done whenever a publication for an object is received. In practice, updating the interest graph is only required immediately proceeding the optimization heuristics, since we care only about the most recent information. Interests also tell us when replicas are stale and can be discarded.

2.5 Publication and Subscription Model

We piggy back several pieces of information on publications so that subscribers can make informed decisions about object migration and learn about servers in a low overhead manner:

node: where the primary for this object resides.
load_stats: Bandwidth load and capacity for the primary's node.

Subscription state is soft so objects must periodically resubcribe to their interests. For simplicity, we assume in our evaluation that each object will make a subscription and a publication each frame. In practice this may not be required in all cases (e.g. some objects that are not highly dynamic, like items, can afford to refresh their replica state less frequently). However, this gives us a good upper bound on resources required for near perfect synchronization (e.g. close to that achieved by servers and clients in current client-server architectures).

2.6 Game Interface

The framework exports an object store interface to the game application. We only require that game objects adhere to the interface shown in Figure 1 for making state changes.

`GetProperty`, `ModifyProperty`, and `ModifySubscription` are implemented by our framework to properly send publications and subscriptions on object state changes. The `type` argument to `ModifyProperty` determines whether the change should be reflected `GLOBALLY` or only `LOCALLY`; this allows servers to tentatively predict changes to replicas, similar to how clients of games *dead-reckon* [18] state changes to avoid delay in object motion, etc.

The `ReceiveUpdate` method is application defined so that it can do update reordering or conflict resolution (for remote updates) if it desires. A simple policy that we believe to be sufficient for most games is first-writer-wins, since client-server interactions behave this way in current architectures.

3 Object Placement Strategies

We explore 3 strategies for placing objects: *static placement*, *region-based placement*, and *load-balanced placement*.

3.1 Static Placement

The most obvious strategy for placing objects is to assign each to a server when they are created (e.g. when a player joins a game) and then leave it there for the remainder of the game. If assignments are initially random, then we expect this to balance the number of objects across servers fairly well. However, we expect this strategy to incur a high publication cost between servers, since objects are likely to be interested in many remote objects.

3.2 Region-based Placement

The primary strategy employed in Massively Multiplayer Role Playing Games (MMORPGs) is to assign a region of the game world to each server and place objects on a server if the object is in that region. If regions are partitioned intelligently, such that objects in one region are not likely to have interests in objects which are in another, then we expect update messages between servers to be minimal. However, objects will move between regions much more frequently in an FPS game, so we expect migration traffic will be high. In addition, regions which are more popular than others will receive increased load, leading to imbalance.

3.3 Load-Balanced Placement

Ideally, we would like to cluster each mutually interested group of objects on a single server, regardless of location or initial server. We describe a strategy that attempts to do that.

Cost Model: For now, assume that each node has capacities $C(in)$ and $C(out)$, all objects represent players, and all objects incur identical costs (we overcome these assumptions below). To simply our model, we denote the cost of an object as a single variable (we expect communication cost to dominate as it is the bottleneck in current game servers). Denote the inbound and outbound costs of a primary as $l_p(in)$ and $l_p(out)$, respectively, and the costs of a replica as $l_r(in)$ and $l_r(out)$.

Each primary *o* must maintain a client connection, maintain its subscription, send publications to R *rendezvous points* (assuming the publish-subscribe model in Mercury), and possibly receive operations (delta updates) from those subscribed to it. Because we expect objects to be mutually interested in each other (if you are in my "area" then I am in your "area"), the primary's host can approximate the set of nodes that are interested in *o*; call this N_o. Hence, we estimate that

$$l_p(in) = client_{out} + |N_o| \cdot |delta| \cdot delta_rate, \quad (1)$$

$$l_p(out) = client_{in} + R \cdot |pub| \cdot pub_rate + |sub| \cdot sub_rate. \quad (2)$$

Similarly, each replica must receive state updates routed to it from the primary and possibly send occasional operations to update the primary. Hence, we estimate that

$$l_r(in) = |pub| \cdot pub_rate, \quad (3)$$

$$l_r(out) = |delta| \cdot delta_rate. \quad (4)$$

In practice, $|delta|$ and $delta_rate$ is so small relative to other costs, that we ignore it in load prediction.

For the constraints we enumerate in the remainder of this section we denote capacity, primary, and replica costs as C, l_p, and l_r, respectively, but what we really mean is that the constraints must hold for both the in and out versions.

Clustering: Given a set of primaries and their interests on a node n, n must decide whether it would be beneficial to migrate a primary o to another node m. Let $I_o(k)$ be the set of objects o is interested in that reside on node k. We will answer positively if $|I_o(m)| > |I_o(n)|$ and $L(m)$, the load on node m, satisfies

$$L(m) \leq L_{highwater} - (l_p + l_r \cdot |I_o(N - \{m\})| + \alpha), \quad (5)$$

where $I_o(N - \{m\})$ is the set of all objects o is interested in not on node m and α is a fudge factor.[1] $L_{highwater}$ is a high water mark for the target node; we do not want to bring the node all the way up to capacity so that it is able to absorbed sudden bursts in traffic. Intuitively, this means we migrate an object if moving it reduces the number of off-node objects it is interested in and it will not overload the node we move it to.

A problem with this formulation is that interests change over time so it may not be accurate to examine interests at a particular instant. To account for dynamic interests, we redefine $I_o(\cdot)$ to be the set of objects o has been interested in for $> T_{stable}$ milliseconds (including brief durations where o is not interested in the object). This approximates the *stable* interests of o (with the assumption that the recent past is a good predictor of the future). We give two informal reasons why this is expected to work: if two players are fighting each other (or with each other, in a team game), they will usually "follow each other around"; if many players hang out in a region, this will capture that interest *if* they hang out in that region long enough for it to matter (it may just be a transient "passing through" point). We force objects to rebuild their interests after migration to avoid frequent movement of the same object.

One possible improvement on this algorithm is to consider primaries as groups, rather than one at a time. By considering more objects at the same time, we can better determine global relationship partitions; however, this is computationally more expensive (there are $\binom{n}{k}$ k-sized groups to consider if there are n primaries on a node), and it will be more difficult to "fit" groups onto other nodes if they are somewhat near capacity.

Load Shedding: This algorithm should function well so long as nodes are not close to capacity, because then they will always be able to accept object migrations that reduce global load. To keep load low, each node will try to evict primaries when load grows above a certain high water mark $L_{highwater} < C$. To decide which primaries to evict, the node will first determine the *connected components* of its primaries based on the graph formed by their interests (this is can be done simply using depth first search in $O(V + E)$ time for a graph $G(V, E)$). Let the set of objects in such a component q be O_q; we can migrate these objects to a node m if

$$L(m) < L_{lowwater} - (l_p \cdot |O_q| + l_r \cdot |I_{O_q}(N)| + \alpha), \quad (6)$$

[1] Since this is already a conservatively high estimate of the modified load — the target node may already be subscribed to some of the $|I_o(N - \{m\})|$ replicas — we set α to 0 in our evaluation, though it is not clear that it would always be beneficial to do so.

where $I_{O_q}(N)$ is the set of all interests of O_q (excluding objects in O_q itself). $L_{lowwater} \leq L_{highwater}$ is a second load threshold; the distance between low and high water marks determines how likely our optimization heuristic will cause oscillation. A small distance means that offloading a component to a node will likely push it over its high water mark, causing it to run this offloading heuristic itself. However, a very large distance means that load may be highly imbalanced since servers are stable and do not attempt offloading in this range. Because they maintain *no* interests on the current node, moving a connected cluster of objects is guaranteed to reduce or maintain global load (with the assumption of stable interests).

It is possible that there is only ever one large connected component per node, in which case partitioning is not possible. In extreme cases it might be worthwhile to run more expensive algorithms such as MIN-CUT to try to partition the interest graph. However we believe that in an FPS game, all such objects are unlikely to be mutually interested for long (since, for example, if so many players are in the same region, a lot of them will either move away or die very quickly and re-spawn elsewhere).

So far we have assumed that objects only learn of other servers through information piggybacked on publications. In this case, it is unlikely that lightly loaded servers in the system will be discovered by heavily loaded ones (since, for example, a server with no load will never exchange any publications). In load shedding, we use the underlying publish-subscribe layer to match up lightly loaded servers with heavily loaded ones. When a server's load drops below the low water mark it publishes itself as a candidate for migration; when a server's load grows above the high water mark, it subscribes to these publications. Instead of multicasting these publications to all subscribers, the rendevous point anycasts each publication to one subscriber so pairwise matches are made.

Details: We assumed that node capacity and object load was static and homogeneous. It is not difficult to see that we can extend our algorithm to deal with dynamic, heterogeneous capacity and load by weighting individual variables. In practice, [10] found that the rate of FPS game traffic is very predictable because game logic deterministically floods updates every 100ms, so static estimates are probably good enough.

There will be non-player objects in the game world (e.g., doors, items, etc.). These do not incur client communication costs and we conjecture are relatively light-weight computationally as well; hence for these objects, we simply remove $client_{in}$ and $client_{out}$ from Equations 1, 2, 3, and 4 and adjust $|pub|$ and $|delta|$ accordingly. We also require different values for T_{stable} for different object types since some objects (e.g., missiles) only for a short period of time and may never form stable interests.

When new objects are created, we must decide where to create them. We assume players initially connect to some server which they discover through out-of-band means; their object will be created there. All other objects that are dynamically generated (like rockets fired from a player's entity) are associated with an *origin* object and are created on the same node as the origin.

4 Implementation

To evaluate our object placement strategy we implemented the object store framework described in Section 2. However, because it would be difficult to implement and evaluate a real game on our system, we instead use a trace driven application. In addition, in order to run controlled experiments, we run the servers on a discrete event based simulated protocol layer. This simulation makes the following assumptions for tractability:

Fixed Routing Cost: The simulator is agnostic to the underlying publish-subscribe system; i.e., we do not model routing details of publications and subscriptions. Load balancing of publication routing is be optimized by the routing layer (i.e., Mercury [4]), so we optimistically assume routing load is equally shared by all servers in the system.

No Propagation Latency: We ignore latency of propagating update and subscription messages. Delay of messages can affect decisions made by entities in the simulation. However, because our trace is based on a sequential play out of a game, we can not model these changes without modifying the trace. We defer study of inconsistency due to delay for another time.

Propagation latency also may effect the convergence properties of our heuristics since oscillations are more likely with stale information. However, we only perform our heuristics on a periodic basis even in simulation so the information it uses is not perfectly up to date (it is up to two frames or 200ms out of date, which is a reasonable bound on delay; higher latency would imply that the game is unplayable anyway).

No Contention or Message Loss: We ignore precise details about server execution speed, cross traffic, queuing delay, message loss, etc. These are details are better measured by real implementation.

In addition, because we assume no message loss or propagation latency, we have not yet implemented a full migration protocol.

5 Evaluation

5.1 Setup

This section describes the workload we used to evaluate our implementation and the parameters we used in our experiments.

5.1.1 Workload

In order to obtain workload data from a multiplayer FPS game we modified Quake 2 [20], a popular FPS game during its prime. We had two goals in modifying Quake 2: the first was to run a match with 128 bots (automated players) in a large game environment, and the second was to instrument Quake 2 to tell us where these bots were located, and what they were doing during each frame.

For each object we log the following each frame: the current frame number, the object's numeric ID, the object's location, the set of objects visible to the object (the interest set) and the set of locations of the interest set (the subscription set). We also log object creations, deletions, and inter-object modifications (e.g. a missile hitting a player).

We classify objects into three categories: players, which are mobile and persistent, items, which are immobile but usually persistent, and missiles, which are highly mobile but exist for a very short time. We filter out or recategorized environmental entities (such as lights, platforms, doors, rotating gears, etc.) because their state is either static (in the case of lights), or because their behavior is the

Workload Trace Statistics		
Number of bots		128
Number of frames		9000
Game time		15 minutes
Map		city64
Total number of objects		793
Avg. number of subscriptions	by players	12.51
	by items	14.83
	by missiles	21.56
Avg. number of interests	by players	16.61
	by items	18.13
	by missiles	29.87

Table 1: Summary statistics of our 128 player botmatch.

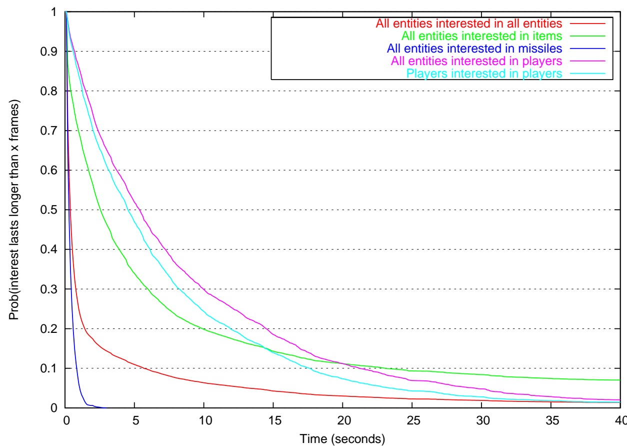

Figure 2: Inverted CDF of interest lengths.

same as items.

A summary of the trace is presented in Table 1. Note that missiles have a higher number of subscriptions and interests than any other class because they quickly travel through a large number of regions during their lifetime, and their interests are not discarded until after they have expired.

Figure 2 shows the inverted cumulative distribution of interest lengths over 40 seconds (400 frames). The probability that any object will be interested in a missile after 3 seconds is zero because the missile does not exist for longer. After 20 seconds items have the highest chance of being in the interest set of other objects. This is probably due to the immobility of items. There is less than 20% chance that one player will be interested in another player for more than 15 seconds. This suggests that bots do not generally chase others around for long periods of time.

Hence, interests in Quake 2 are *highly dynamic*, and this workload should stress our object placement strategies.

5.1.2 Message Cost Estimation

We estimated fixed sizes for different message types using Quake 2 as a guide (shown in Table 2). Publications are estimated to be 4 times the average size of object state sent to clients each frame.

type	*pub size* (B)	*sub size* (B)	*migrate size* (B)
player	64	$4S$	741+$4S$
item	16	$4S$	25+$4S$
missile	64	$4S$	41+$4S$

Table 2: Estimated cost of messages. S is the number of subscriptions currently held by the object.

We inflate the size because we expect modern games to require more state, and because servers may want to maintain higher fidelity replicas (e.g. a critical component of state exchange is extra information used for local prediction, like an object's velocity). An objects expresses interest using the BSP tree nodes [11] in its *potentially visible set* (PVS), each of which is an integer value. This is the most strait forward mechanism to express subscriptions in an FPS, since the map is partitioned using a BSP tree and the PVS is already precalculated. Objects publish themselves and send a subscription every frame (100ms). In practice, this is not required since state may not change every frame and some objects (e.g. items or objects that are "far away") do not require timely high fidelity updates. However, this gives us an estimate of the cost in an untuned application.

A remote update is generally 8 bytes plus protocol overhead (15 bytes total) because the update is almost always a missile impacting a player (modifying its health value) or a player picking up an item (toggling a flag on the item).

Costs for migrating objects are estimated as follows: For players, we use the size of the client and entity state structures (minus variables which were clearly used for scratch results, only used on the client, or function pointers and constants). For items and missiles, we estimated the required state by hand because most of the data structure used to represent them is irrelevant (Quake 2 uses the same data structure for all object types). In addition, we add overhead for these messages.

We do not add IP or transport layer overhead to our measurements because in a real implementation, all the messages between two servers sent within a frame period would be delayed and aggregated together (it is unlikely that all of them exceed 1500 bytes). Since we only use 10 servers in our simulation, the maximum possible UDP/IP overhead per server is $\frac{2\times10\times28\text{B}}{100\text{ms}} = 5.6$KBps[2] and would not impact our relative results.

We estimate inbound and outbound bandwidth from each client as constant rates of 3KBps and 4KBps respectively from measurements in [10]. We assume routing load (cost of forwarding publications and subscriptions) is equally shared among all servers.

5.1.3 Simulated Environment

We simulate 10 servers and assume that all players in the game can be served equally well by any of the 10. Though we did not explicitly cap server capacity, we found that this setup could be supported with symmetric 1.5Mbps links (which is admittedly an optimistic assumption for outbound bandwidth in the United States, but not completely unrealistic in the future).

[2]The factor of 2 is for being a routing hop.

type	T_{stable} (ms)	$T_{discard}$ (ms)
player	1500	2000
item	1500	3000
missile	never	always

Table 3: Stability and discard values for objects.

Unless otherwise stated, for the load balanced strategy, we used high and low water marks of 140KBps and 90KBps (excluding routing cost), respectively, and considered migrating each object (i.e., ran our heuristics) once every second. Servers estimate their bandwidth usage by using a bucketized 1 second moving window. Based on workload statistics (see Section 5.1.1), we estimate the T_{stable} values for each object type, in addition to the amount of time before stale interests and replicas are discarded (see Table 3). We never keep objects' interests in missiles since they never exist for more than 2 or 3 seconds.

We simulate the static placement policy by intially placing objects on servers intelligently to balance the client load. Other intial objects are assigned randomly.

We simulate a region-based placement policy by "learning" clusters of BSP tree nodes from the trace (which lists all the potentially visible nodes for each object position during the game). Briefly, we chose several initial disjoint clusters and then grew them by greedily adding nodes in subscriptions that overlapped existing clusters; the clusters were then assigned to servers in a load balanced fashion.[3] We believe this is a relatively good partitioning of regions since it reduced the migration frequency by a factor of 5 compared to a random assignment of BSP tree nodes.

5.2 Results

We simulated the execution of the Quake 2 trace using each of the object placement policies. This section evaluates how well each policy estimates dynamic object interests in an actual playout, and approximates the costs of each policy if the game were actually played on a distributed architecture. We use two primary metrics to evaluate the strategies: communication cost and migration frequency.

5.2.1 Instantaneous Communication Cost

This metric demonstrates how well each strategy manages load at any given point in time. In addition, it shows the impact of dynamic communication patterns such as bursty traffic.

Figure 3(a) shows the average bandwidth used by servers for each 500ms time period during the trace. It is clear that the static placement fares the poorest, using almost double the bandwidth as the region-based and load balanced policies. This is in line with our hypothesis that a static placement would be expensive due to large numbers of remote interests (resulting in many exchanges of publications). The region-based and load balanced lines are similarly centered, though it appears the region-based average is more stable over time. This is to be expected, since the population size of a particular region in the map is likely to remain stable over time (e.g. only a finite number of players will "fit" comfortably in a region), while the load balanced policy does not necessarily heed region boundaries. Nevertheless, we see that workload dynamics cause bursts in traffic for all three policies (for example, the spike in traffic at about 150 seconds). The outbound cost is a little higher than inbound because clients consume 4KBps and only produce 3KBps.

Figure 3(b) shows the minimum and maximum bandwidth used by any server in each 500ms time period. As predicted, the static placement policy results in very balanced load, with max and min load differing by at most 20KBps at any given time. Of course, this just means every sever is heavily loaded. Interestingly, the maximum outbound bandwidth for the region-based policy is higher (sometimes much higher) than the load-balanced policy. We see later that this is most likely due to an overestimate of subscription costs (see Section 5.2.2).

Both the region-based and load balanced policies appear to incur imbalanced load; the maximum inbound and outbound loads are about 150KBps and 170KBps respectively, while the minimums are substantially lower, both under 100KBps. Further, the minimum usage for the load balanced policy spikes at many points in the simulation. We will see that the load for the load balanced policy is not actually imbalanced below; this is only the case when examining a small time slice — over time load balances out. In fact, this is why we observe spikes in the minimum load: spikes occurs when highly loaded servers offload clusters of objects to the lightly loaded servers. The bimodal distribution of instantaneous load is, in fact, a blessing rather than a curse. Because there are always one or two lightly loaded servers (below the low water mark), while the others are at "normal" load (between low and high water marks), there are always candidate servers for offloading when bursts occur. We observed that when all servers are allowed to go near or above high capacity, their load never drops off again (this can be seen in the static placement case). Hence, maintaining a set of lightly loaded servers is important to the operation of our heuristics.

5.2.2 Aggregate Communication Cost

This metric demonstrates how well each strategy balances load over time.

Table 4 shows bandwidth statistics aggregated over the duration of the 15min trace. We see that over time, the load is fairly well balanced in the load balanced case; for inbound bandwidth, the balance is as good as the static case. However, the same can not be said for the region-based policy; the instantaneous load imbalance seen in Figure 3(b) is aggregated over time. This supports our hypothesis that certain regions are always more popular than others, causing higher load on the servers responsible for those regions. One might consider whether we could maintain load balance by off-loading responsibility for regions when a server becomes imbalanced (as is done in Butterfly.net [5]). This is not guaranteed to reduce load however; some servers are only assigned one very large region, so they would have to partition the region to offload it. There will probably be many objects on one side of the partition interested in the other, and visa versa.

Figure 4 shows the breakdown of aggregate bandwidth usage by

[3] The minimum and maximum number of nodes per server differed by about a factor of 2; though this does not translate into how "large" a region is — a BSP tree node is just half of a space partition.

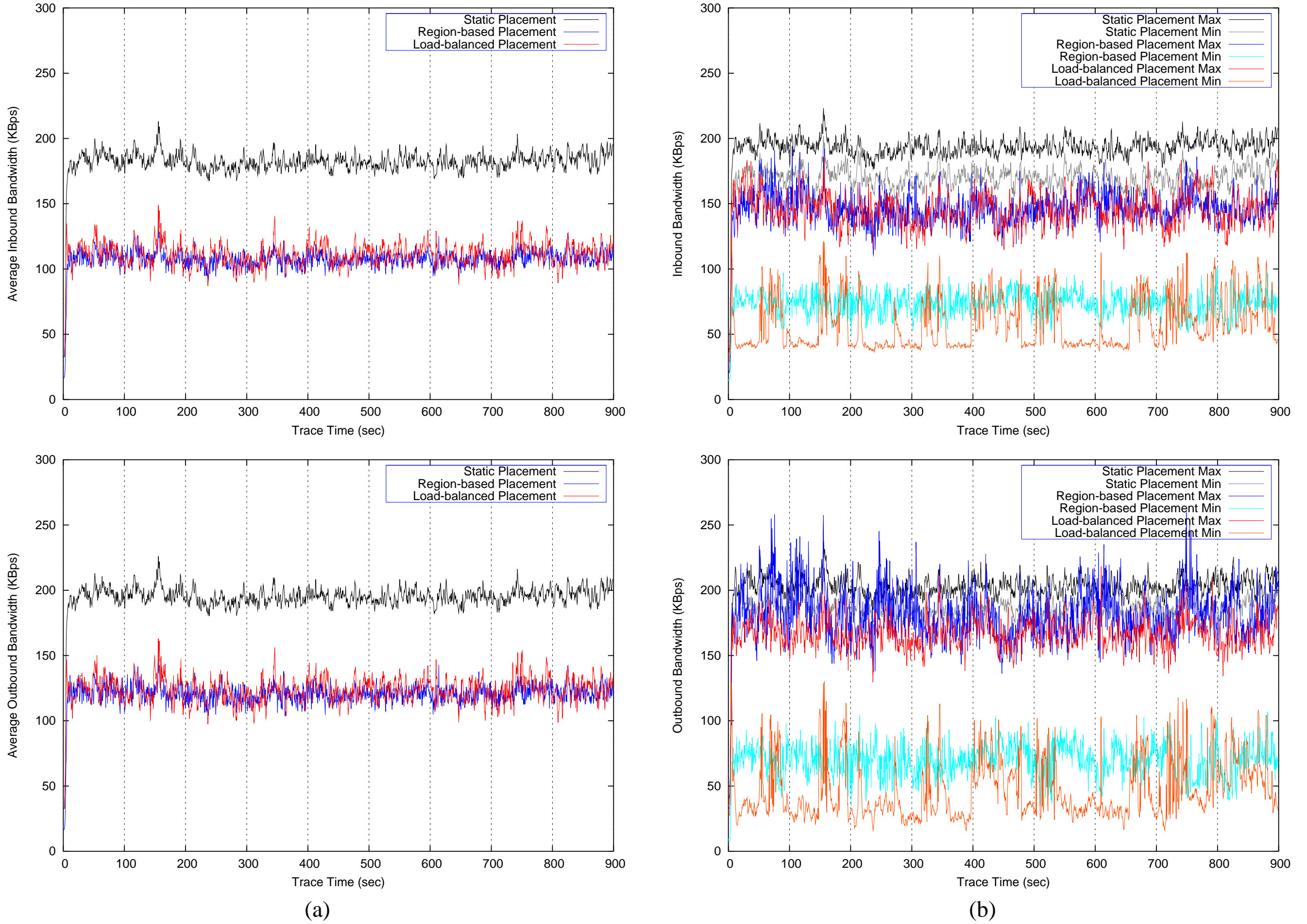

Figure 3: (a) Average bandwidth usage over each 500ms period for each strategy; (b) Minimum and maxium bandwidth usage at any server over each 500ms period for each strategy.

message type for the average and maximally loaded servers. Here we see that the primary cost of the static strategy is from receiving publications (which are routed at the rendezvous points and hence counted as outbound routing bandwidth). We also see that the maximally loaded server with the region based policy incurs most of its outbound cost from subscriptions; this is due to the fact that we do not aggregate subscriptions over different objects in our simulation and the result is magnified by the fact that objects in the same region will have identical subscriptions. Hence the cost of subscriptions, especially for the region-based case will be lower in a real implementation. One might wonder if some publications or subscriptions are even necessary in the region-based case, since a server responsible for a region is guaranteed to have all the primary objects for that region. From this breakdown we see this optimization would yield minimal benefit since servers still receive more publications than they send (implying that there are still many inter-region interests).

We also observe that the cost of migration only begins to impact the region-based case. Because it occurs rarely compared to publications, it is not a significant factor in the total cost. In addition, updates are very small and less frequent than publications, so their impact is negilable. Finally, we note that in the static case, client load is perfectly balanced, while in the load balanced case, the maximally loaded server differs from the average by approximately one client, and in the region-based case, the maximally loaded server differs from the average by approximately two clients.

Table 4 also compares bandwidth usage with an estimate of using a single server to host all the players. In this case, the only bandwidth costs are for client connections. 384KBps-in/512KBps-out is obviously substantially more than we could provision at a single point in our target environment (and with 128 players, the computational cost is probably also prohibitive), though the global resource usage by the distributed architecture is almost a factor of 3 more expensive. However, note that we have assumed that in the distributed case all objects naively publish their state every frame even if no remote objects are interested in them. This design decision allows new subscribers to receive publications without explicit knowledge of the publisher and visa versa; this is important to maintain scalability and fault tolerance in a decentralized distributed system. In addition, in our rendezvous based publish subscribe system (another design decision made for scalability), each publication and

	Inbound Bandwidth (KBps)					Outbound Bandwidth (KBps)				
	avg	*stddev*	*min*	*max*	*total*	*avg*	*stddev*	*min*	*max*	*total*
Static Placement	182	5	175	192	1819	195	4	188	201	1947
Region-based Placement	107	20	78	142	1074	120	29	75	171	1204
Load-balanced Placement	111	5	100	119	1109	124	10	104	138	1235
Single Server Placement	38	36	0	384	384	51	49	0	512	512

Table 4: Bandwidth statistics for each strategy aggregated over the entire 9000 second trace period. Single Server Placement estimates resource usage if it was possible to place all 128 players on one of the 10 servers.

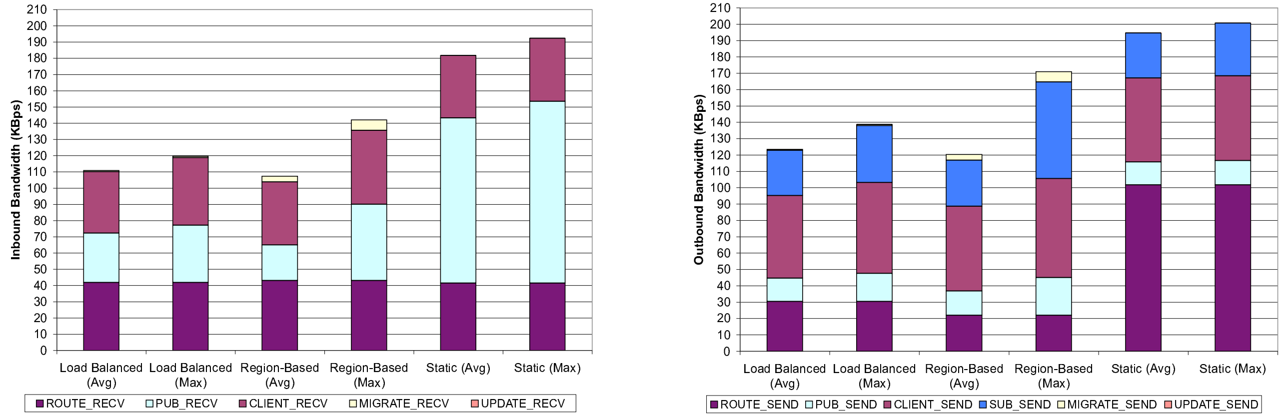

Figure 4: Breakdown of bandwidth costs on average and for the maximally loaded server with each placement policy.

subscription is subject to at least one routing hop.[4] Hence, mandatory costs of scalable distribution account for much of the overhead. Finally, we note again that our estimates for publication and subscription rates and sizes were very conservative and a tuned application would likely do much better.

5.2.3 *Migration Frequency*

In a real life scenario, latency is likely to be as large a concern as bandwidth usage (if not larger, for playability reasons). Because we have not yet determined how disruptive an object migration would be to participants in a game (though we believe various prediction techniques can be used to mitigate the impact), we relatively score the three placement strategies based on how often they migrate objects. However, we note that in a distributed architecture, migration is not always a "bad" thing, since moving a primary closer to its interests will ensure better update consistency.

Figure 5 shows how often an object of each class migrates over each 10 second time period. For example, the rate is about 0.5 for players with the load balanced policy, so on average we would expect a player to migrate once every 20 seconds. Items do not migrate in the region-based policy because they cannot move, and missiles do not migrate in the load balanced policy because they do not exist long enough for interests to become stable.

As expected, the migration rate is high for the region-based policy. In particular, the rate of migration for missiles (more than once every second on average) is troubling since they only *exist* for a few seconds; if we were to actually employ this policy, we certainly would need to special case missiles to prevent migration. Nonetheless, the migration rate for players in the region based case is still almost 4 times as high as the load-balanced case and has higher variance. Thus we are inclined to believe that our clustering heuristics do in fact capture interests beyond those based solely on regions.

However, a migration rate of once each 20 seconds is still rather high for player objects since the migration of a player may be disruptive. We may explore the impact of providing quality of service guarantees in the future (e.g. only migrate players when they die, are not at "critical" points in the game, etc.). However, because of the highly dynamic and transient nature of interests in our FPS trace, it is unlikely that a lower migration rate could achieve as low a load. In other game genres, interests are probably much more stable and our heuristics would be more applicable.

5.3 Discussion

We conclude that from the perspective of bandwidth consumption and migration frequency, the load balanced policy does as well or

[4] In the case of 10 or 100 servers, this is probably only one hop, since pointers can be cached.

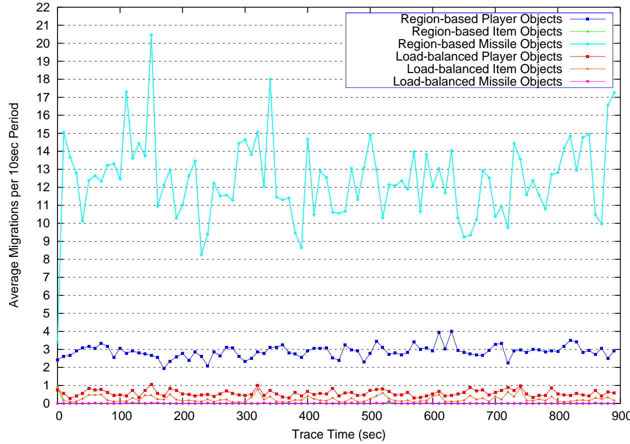

Figure 5: Average migration rate over each 10 second time period for each of the object classes.

better than the static and region-based ones, but there is much that can be improved. If we run our heuristics at rate slower than once a second, the benefits of optimization fall off rapidly (and degenerate to the static case if we only run them every 5 seconds). This is primarily due to the fact that once the load on all the servers grows above the low water mark, there are no candidate servers to offload clusters to. After the load grows above the high water mark, we can not even perform the clustering optimization due to fear of overloading the target. We believe the solution to this problem is to use dynamic water marks based on an estimated value of global load instead of static ones; if global load is high, then each server should be more willing to receive offloaded clusters and we should be more aggressive about optimizing interests to reduce the number of total replicas in the system. Due to the difficulty of approximating how a distributed global load estimator would function, we did not include one in our simulation. However, the Mercury publish-subscribe system has precisely such a mechanism, and we can use it as a bootstrap.

In addition, since we must run the heuristics so often, its computational cost becomes a concern. In our untuned implementation, we observed that the actual interest graph traversals are relatively cheap (on the order of a couple milliseconds), but actually updating interests for every primary object in the store takes about 150ms for a store size of about 200 objects on a Pentium 4 2.4Ghz with 1024MB RAM. We estimate that a tuned implementation of the store (using an auxiliary array for iteration instead of iterating over an associative list) would yield a factor of 10 improvement. Furthermore, if we incorporate game semantics, we can eliminate this overhead all together since we can bucket objects by their BSP tree nodes and use the precalculated PVS to determine interests. This would tradeoff generality for performance.

Although we have evaluated two broad characteristics of three object placement strategies for distributing a FPS game across multiple servers, we have so far ignored other properties such as fault tolerance and cheating prevention.

An argument can be made that the load balanced strategy is able to contain faults better than a region-based policy because a server is only required for the continued operation of the object it maintains; the other objects in the game can continue functioning fully even if some servers fail. In a region-based scenario, a portion of the game world can not be entered when a server fails and will remain unavailable until either the responsible server recovers or another server picks up the regions the failed server was responsible for. In addition, in a region-based scenario, servers must all agree upon who the owner of a particular region is. This either implies centralized management, state which is globally replicated and synchronized, or use of a distributed location network like a distributed hash table (such as was done in [16]). Each of these approaches has disadvantages either in terms of scalability or routing delay. Note that because of our assumption of a rendezvous based publish subscribe system and state publications on every frame, we always have an up-to-date partial replica backup at the rendezvous point. Whether this backup would be sufficient as a fail-over depends on game semantics.

6 Related Work

We modeled our problem as a graph cutting problem. Dahlhaus, et al. [8] showed that this general problem is NP-Hard when we have three or more clusters. Several approximation algorithms exist [8, 21, 14], but our problem is much more complex since: (1) each node must decide where to place objects with only local (incomplete) knowledge, otherwise our system can not scale, (2) object interests change quickly and (3) we must deal with the complexities of heterogeneous systems.

The Sprite operating system [9] migrated processes to offload work to idle machines. Migration is much cheaper in our system due to smaller object size and there is a higher degree of dynamic interaction between objects, so we can afford to perform migration much more frequently. Emerald [15] implements mobility primitives for migrating fine-grained objects. Although most migration decisions are made by the programmer in Emerald, we leverage their object design in our system.

Abacus [1] automatically migrates objects between server and client machines by monitering their black-box operation. Although we leverage some statistics collection mechanisms, our environment is much less structured. In addition, Abacus focuses on maximizing application performance, while our main goal is to reduce communication cost (with constraints on acceptable performance).

The High Level Architecture (HLA) is a specification for a common technical architecture for modeling and simulation. The HLA uses a publish-subscribe model, and object updates are performed at the data distribution management (DDM) layer by using a broadcast-based scheme [3]. Although some research has been done to limit publication and subscription cost [23, 3, 17], to our knowledge no work has been done in migrating objects in HLA.

Many multiplayer games use a centralized server to manage game state and updates, where all of the objects in the system are located at the server, and replicated to the players in the game. In order to reduce the number of updates sent over the network, area of interest filtering is done at the server [22]. Games such as Quake [19] and CounterStrike [6] behave in this manner.

Another strategy used in multiplayer games is parallel simulation.

Each player in the game simulates the entire game world, and thus, each object is replicated at every node. Parallel simulation has advantages for cheating detection and prevention, though object updates must be multicast to all participating players and they must move in lock-step. Microsoft's Age of Empires [2] is a canonical example of a multiplayer game employing parallel simulation.

Finally, games can have their state fully distributed among players. miMaze [12] was the one of the first fully distributed multiplayer games, and it had each player compute it's own local view of the global game state based on the updates it received from other players in the system. However, there was no notion of authority or ownership of objects in the system. Knutsson, et al. [16] proposed distributing a MMORPG among players using a distributed hash table, but their evaluation focused on fault tolerance.

Companies such as Butterfly.net [5], TerraZona [24] and Global Gaming [13] are developing middleware to make MMOG development simpler. Butterfly.net is implemented on a Grid cluster, and uses a replica migration model similar to our region-based migration policy. TerraZona provides software infrastructure for game developers but also partitions the world by region. Global Gaming distributes game state among the players in a game. Our target environment is neither a central cluster or pure peer-to-peer; we utilize a community of decentralized servers, which have much tighter constraints on communication.

7 Summary

We have presented two optimization heuristics for placing objects in distributed multiplayer games based on *interest* management: clustering and load shedding. We showed that these heuristics achieve relatively low communication cost, load balance, and a lower migration frequency than other placement strategies. In addition, we evaluated several intrinsic properties of the play-out of a real FPS game and showed that interests are highly dynamic and transient. Hence, we believe that our heuristics would perform even better on games with more stable object interests.

One important characteristic that we believe should be evaluated in the future is how migrations impact perceived performance in a game. Our heuristics hinge on the ability to aggressively optimize placement in highly dynamic environments, so it would be useful to know what constraints must be placed on migration in order to achieve acceptable client performance.

Regardless, we have demonstrated that interest management is a useful method for inferring relationships between objects in multiplayer games, and believe it can be useful in other contexts as well.

References

[1] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the USENIX Annual Technical Conference*, pages 307–322, 2000.

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

[3] N. T. Azzedine Boukerche, Amber J. Roy. Dynamic grid-based multicast group assignment in data distribution management.

[4] A. R. Bharambe, S. Rao, and S. Seshan. Mercury: a scalable publish-subscribe system for internet games. In *Proceedings of the first workshop on Network and system support for games*, pages 3–9. ACM Press, 2002.

[5] Butterfly.net. http://www.butterfly.net.

[6] CounterStrike. http://www.counter-strike.net.

[7] AmdZone, "Valve releases hammer port of counter-strike server". http://www.amdzone.com/.

[8] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiway cuts (extended abstract). In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 241–251. ACM Press, 1992.

[9] F. Douglis and J. K. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[10] W. Feng, F. Chang, W. Feng, and J. Walpole. Provisioning online games: A traffic analysis of a busy counter-strike server. In *Proceedings of the Internet Measurement Workshop*, Nov 2002.

[11] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. earson Addison Wesley, 1990.

[12] L. Gautier and C. Diot. Design and evaluation of mimaze, a multi-player game on the internet. In *International Conference on Multimedia Computing and Systems*, pages 233–236, 1998.

[13] Global Gaming. http://www.global-gaming.com/.

[14] K. Hogstedt, D. Kimelman, V. T. Rajan, T. Roth, and M. Wegman. Graph cutting algorithms for distributed applications partitioning. *ACM SIGMETRICS Performance Evaluation Review*, 28(4):27–29, 2001.

[15] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[16] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM 2004*, 2004. To appear in.

[17] T. Lu, C. Lee, W. Hsia, and M. Lin. Supporting large-scale distributed simulation using hla. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 10(3):268–294, 2000.

[18] L. Pantel and L. C. Wolf. On the suitability of dead reckoning schemes for games. In *Proceedings of the first workshop on Network and system support for games*, pages 79–84. ACM Press, 2002.

[19] Quake. http://www.idsoftware.com/games/quake/.

[20] Quake 2. http://www.idsoftware.com/games/quake/quake2.

[21] T. Roxborough and A. Sen. Graph clustering using multiway ratio cut. In G. Di Battista, editor, *Proc. 5th Int. Symp. Graph Drawing, GD*, number 1353, pages 291–296. Springer-Verlag, 18–20 1997.

[22] J. Smed, T. Kaukoranta, and H. Hakonen. A review on networking and multiplayer computer games. Technical Report 454, Turku Centre for Computer Science, Apr. 2002.

[23] G. Tan, L. Xu, F. Moradi, and S. Taylor. An agent-based ddm for high level architecture. In *Proceedings of the fifteenth workshop on Parallel and distributed simulation*, pages 75–82, 2001.

[24] Terazona. http://www.zona.net.