REED: Optimising First Person Shooter Game Server Discovery using Network Coordinates

GRENVILLE ARMITAGE and AMIEL HEYDE

Centre for Advanced Internet Architectures

Swinburne University of Technology, Australia

Online First Person Shooter (FPS) games typically use a client-server communication model, with thousands of enthusiast-hosted game servers active at any time. Traditional FPS server discovery may take minutes, as clients create thousands of short-lived packe t flows while probing all available servers to find a selection of game servers with tolerable round trip time (RTT). REED reduces a client's probing time and network traffic to 1% of traditional server discovery. REED game servers participate in a centralised, incremental calculation of their network coordinates, and clients use these coordinates to expedite the discovery of servers with low RTTs.

---

1. INTRODUCTION

Internet-based multiplayer First Person Shooter (FPS) games (such as Quake III Arena, Counter-Strike:Source and Team Fortress 2) have become quite popular in the past decade. They commonly operate in a client-server mode – players control *game clients*, on a personal computer (PC) or dedicated games console, that communicate with *game servers* hosting individual games. Game publishers rely on enthusiasts – Internet service providers (ISPs), dedicated game hosting companies and private individuals – to independently host the tens of thousands of actual game servers active on the Internet at any time [Feng and Feng 2003].

A challenge for game clients is *server discovery* – often a manually-triggered process of locating up-to-date information about active game servers so players can select a suitable server on which to play. PC-based FPS games traditionally utilise a 'rendezvous' service – active game servers register themselves with a *master server* (often provided by the game publisher), and clients query the master server for a list of currently registered game servers [Henderson 2002]. From the master server's list a client then probes each game server in turn for information such as the network-layer round trip time (RTT), what game is currently playing, and the number of current players. Players consider all this information, presented via the client's on-screen *server browser*, then choose one game server to join.

Some console-based games (such as XBox Live's Ghost Recon and Halo 3 [Lee et al. 2008]) utilise a central 'matchmaking' service. With XBox Live, after configuring their console to play a particular type of game, a subset of prospective players will find that XBox Live has temporarily assigned their console to be a game server for that type of game. The consoles of other players interested in playing the same

game type will receive a list of IP addresses of current game servers from XBox Live, actively probe available game hosts to determine RTT, and allow their players to select an appropriate game server to join.

Competitive online FPS game play usually requires $RTT \leq 150ms \sim 200ms$ [Armitage 2003; Beigbeder et al. 2004]. Clients may spend minutes probing thousands of game servers to satisfy a player's desire to select from a handful game servers with acceptably low RTT. As each probe creates a new, short-lived network-layer flow, server discovery can also temporarily overload devices that keep per-flow state (such as wired or wireless home gateways doing network address translation, NAT).

Our challenge is to minimise the probing a client performs when locating a set of playable game servers under the rendezvous service model. Our solution is REED[1] – a novel application of previous work on virtual *network coordinates* [Ledlie et al. 2006; Dabek et al. 2004] in order to significantly reduce the time a player spends doing server discovery, minimise a game client's network traffic and limit per-flow state imposed on network devices near the client.

Network coordinates are an artificial construct, where the euclidean distance between the coordinates of any two hosts approximates the actual RTT between them. Under REED the master server calculates, and regularly recalculates, network coordinates for each registered game server. A REED client initialises its position in coordinate space relative to a small handful of game servers, then asks the master server for a list of all remaining game servers ranked by increasing 'distance' from the client's coordinates. By walking this list in order, the client probes game servers in roughly ascending RTT and, therefore, can terminate the process early once sufficient game servers with acceptably low RTT have been probed.

We illustrate our approach using Valve Corporation's Counterstrike:Source (CS:S) [Valve Corporation 2009a], and show that clients can discover playable game servers with as little as 1% of the network resources and time used in traditional server discovery. REED may be more generally applied to any FPS game that utilises the rendezvous model of server discovery.

Our paper is organised as follows. Section 2 illustrates the challenges of current FPS server discovery and discusses prior related work, while Section 3 introduces network coordinates. REED is described in Section 4 and analysed in Section 5. Sections 6 and 7 summarise performance issues and future work, and we conclude in Section 8.

2. DISCOVERING PLAYABLE SERVERS – A CHALLENGE

Finding playable FPS game servers in a timely manner with minimal network load is a well recognised problem [Claypool 2008]. Players who trigger server discovery do not have the identity of any particular game server(s) in mind. Rather, they hope to discover a reasonable selection of nearby (low RTT) game servers, from which they can manually choose one based on additional characteristics such as a server's current game type, map type, number of current players, etc.

In this section we use Valve's Steam game management system [Valve Corporation 2009d] and CS:S to illustrate the challenges of traditional server discovery, then summarise some prior attempts to optimise the server discovery process.

---

[1]Easy to pronounce, and happens to be the acronym for "RTT Estimation to Expedite Discovery".

Algorithm 1 Steam's client-side server discovery process over UDP/IP

---

(1) Send a *getservers* query to master server at `hl2master.steampowered.com`:27011

(2) Receive a *getserversResponse* packet containing the <IP address:port> pairs of up to 231 active game servers (in no particular order)

(3) Send one *A2S_INFO Request* probe packet to each game server in the *getserversResponse* list, eliciting an *A2S_INFO Reply* packet from every active game server

(4) Repeat 1, 2 and 3 until the master server has no more game servers to return

---

2.1 Valve's CS:S Server Discovery – an illustrative example

Active, public CS:S game servers register themselves with Steam's master server so they may be discovered and probed by CS:S clients. In mid-2009 there were over 36 000 CS:S game servers world-wide to chose from at any given time of day.

    2.1.1 Registering with the master server. Game servers (re-)register with the master server every five minutes by sending a short, UDP/IP 'join' request, receiving an acknowledgement with 'challenge' code, then replying with brief details[2] of themselves [Valve Corporation 2009b]. When shutdown gracefully, game servers send a 'quit' message to the master server to de-register. Failure of a game server to regularly re-register is treated as an implicit 'quit'. Registered game servers are included in the list of servers reported to CS:S clients who query the master server.

    2.1.2 Client-side behaviour. A player triggers Algorithm 1 when they initiate server discovery through their Steam client's built-in server browser [Valve Corporation 2009b; 2009c]. Outbound *A2S_INFO Request* UDP/IP packets are 53 bytes long and inbound *A2S_INFO Reply* packets (containing server-specific information such as the server's current map, game type, number of active players and so on) average 135 bytes or more. Each game server's RTT is estimated from the time between sending an *A2S_INFO Request* and receiving the matching *A2S_INFO Reply*. The on-screen list of available game servers (often ranked by RTT) is updated as replies arrive and new RTT and server-specific information becomes available. Players ultimately use this information to select and join one game server.

2.2 Traditional server discovery consumes significant time and network resources

Traditional game server discovery can take many minutes, and potentially disrupt operation of NAT-enabled devices such as gateways and wireless access points.

    2.2.1 Speed of server discovery. Probing thousands of game servers takes a noticable amount of time. A Steam client caps its emission rate of *A2S_INFO Requests* based on its apparent network connection speed as indicated by the player (the client offers settings such as '*Modem - 56K*', '*DSL > 256K*' or '*DSL/Cable > 2M*'). For example, a client configured for '*DSL > 256K*' emits roughly 140 probes per second and would take about 257 seconds to probe 36 000 game servers.

    A player cannot simply overstate their network connection speed to enjoy higher probe rates. Counterproductively, this can cause congestion on the player's uplink,

---

[2]The challenge code, game server name, current game type, current map, self-reported geographic region, any requirement for passwords before players can join, and so on.
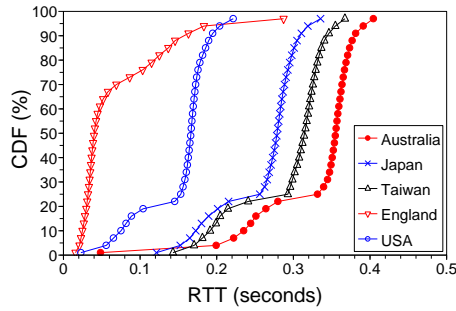
Fig. 1. Clients in five different countries see quite different distributions of measured CS:S game server RTTs
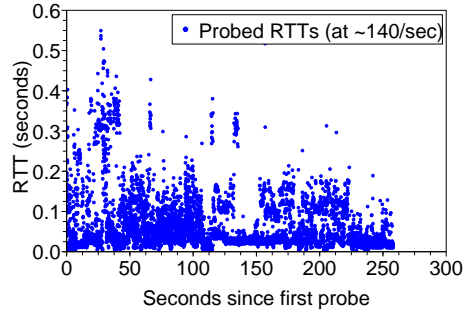
Fig. 2. RTTs observed by an English CS:S client probing 36K game servers (mid-2009). The probe sequence is unrelated to observed RTTs

resulting in inflated RTT estimates and/or dropped (lost) probe packets.

2.2.2 *Short-lived UDP flows can be problematic.* Every UDP probe represents a new, albeit short-lived, network layer flow. Devices that keep per-flow state (such as NAT-enabled home gateways, or wireless hot-spot access points) usually allocate memory proportional to the number of flows considered currently active. UDP flows lack an explicit end-of-flow indication, so memory allocated for UDP flow state is often not released until the flow has been idle for multiple minutes. Conventional server discovery traffic can exhaust the memory of such devices, briefly rendering them unable to forward new traffic (to or from anyone) until earlier flows are declared idle and the associated flow state entries released.

2.2.3 *All game servers are probed to find playable ones.* Game servers are not distributed uniformly around the planet, and the master server returns game servers in no particular order. Consequently a client must probe all game servers before presuming to have seen all (or even many) of the servers close to the client's location.

For example, in mid-2009 there were many CS:S game servers in Europe, a moderate number in the USA and few in Asia. Figure 1 reveals how clients in five countries perceived this distribution of CS:S game server RTTs. Finding servers with $RTT \leq 150ms \sim 200ms$ is rare for Asia-Pacific clients but common for clients in England (and European clients more generally).

Figure 2 shows the RTTs actually measured by an English client versus time since the first probe (at 140 probes/second). The order of probing is clearly not driven by any a priori knowledge of the RTTs likely to be experienced by the client. CS:S clients in all regions experience this limitation, so clients far from the majority of game servers (such as Figure 1's Asia-Pacific clients) end up probing many game servers that, realistically, are unsuitable for competitive online play. Simply probing the master server's first few hundred game servers would reduce a client's network traffic, but provide an unsatisfactory selection of game servers to the player.

From the game server perspective, registered game servers are regularly probed by thousands of potential players (and some automated monitoring systems such as http://www.serverspy.net/). As players probe *before* selecting a game servers for play, both popular and unpopular (or idle) game servers attract similar levels

of probe traffic (easily Gbytes over many weeks [Zander et al. 2005]), regardless of how far (in RTT) the players are from each server.

2.2.4 *Filtering at the client or master server.* Steam allows for client-side filtering to simplify a player's decision process (such as not showing full or empty servers after they've been probed), but this does not reduce the number of probes sent.

A Steam client may also request only game servers of a certain type (such as "only CS:S game servers") or servers who have self-reported to be in one of eight geographical regions (such as "US-West", "Europe", etc). The master server returns a filtered subset of registered game servers, and the client sends fewer *A2S_INFO Request* probes. However, this coarsely-grained geographic selection is only a rough guide to probable RTTs, and still leaves the client probing thousands of game servers in no particularly useful order.

2.3 Related and prior examples of expedited FPS server discovery

Matching clients to game servers with suitably low RTTs may be fully or partially automated – for example, re-locating clients to optimally placed servers [Chambers et al. 2003], or a matchmaking service automatically assigning one client to act as the temporary game server for nearby clients [Lee et al. 2008]. However, rendezvous-based FPS server discovery involves players who wish to be presented with, and personally select from, multiple active game servers having suitably low RTT. We cannot simply auto-assign a player to the lowest RTT game server.

REED is not the first attempt to ensure closer game servers are probed and presented to players before more distant servers. The following scheme was envisaged in [Armitage et al. 2006] and articulated in [Armitage 2008a; 2008b]: A client first retrieves all registered game servers and arranges them into *clusters* likely to have similar topological distance from the client (for example, game servers falling under a common IP address prefix and countrycode (CC) [Armitage 2008a] or Autonomous System (AS) number [Armitage 2008b]). A representative RTT to each cluster is then established by probing a handful of game servers from each cluster ('calibration'). This client then performs 'optimised probing' by ranking clusters in order of ascending RTT and probing all remaining game servers according to their cluster's rank. Clients located far from most game servers (such as Asian CS:S clients) had their probe traffic and probing time reduced to less than 20% of non-optimised server discovery. However, clients close to the majority of game servers (such as European CS:S clients) saw basically no improvement.

Clustering has many weaknesses. Tens of seconds may pass before calibration begins, as the client first retrieves all active game servers from the master server. There may be thousands of calibration probes, proportional to the number of clusters rather than the number of game servers. (For example, in [Armitage 2008b] clustering on /16 address prefixes within each AS resulted in $\sim 1K$ clusters and $\sim 3K$ calibration probes for CS:S.) A few thousand active servers spread uniformly among countries or ASes will generate far more calibration probes than a few thousand servers in a handful of countries or ASes. Finally, there is a dependency on third-party information. Clustering by country requires that either clients or master server contain regularly-updated IP address geolocation mappings (such as MaxMind's GeoLite Country database, `http://www.maxmind.com/app/geoip_`

`country`). Clustering by AS requires IP address to AS mappings in the master server, updated regularly from the Internet's inter-domain routing infrastructure.

REED is not the first to utilise network coordinates for online game server discovery. In 2009 Htrae [Agarwal and Lorch 2009] demonstrated that a novel synthesis of geolocation and Earth-like network coordinates could improve the peer-to-peer matchmaking process used by an XBox Live game (Halo 3). XBox Live assigns certain game consoles (peers) to be both client and (temporarily) a server for up to 15 other game client peers. Htrae uses third-party mappings of IP address to latitude and longitude to 'geographically bootstrap' the calculation of network coordinates for new game consoles. Each peer's coordinates are refined through subsequent inter-peer probing (clients seeking suitable game servers). Htrae's improved matchmaking performance (relative to other schemes) crucially relies on the geographic bootstrapping of peer coordinates for initial estimates of RTTs between peers.

In contrast, REED significantly improves rendezvous-based server discovery. REED clients use two orders of magnitude fewer calibration probes than CC- and AS-based clustering approaches, regardless of how many game servers are currently registered, or how they are topologically distributed. REED also does not rely on third-party information to identify clusters of game servers with similar RTT or to bootstrap its coordinate-based RTT estimation.

3. NETWORK DISTANCE ESTIMATION

Our goal is to minimise the number of IP packets a client (IP_x) must send or receive in order for it to rank multiple game servers (IP_y) by ascending RTT, and then probe them in rank order for additional server-specific information. *Indirect estimation* and *embedded network coordinates* have emerged in the past decade as two classes of techniques for an endpoint IP_x to estimate its RTT to endpoint IP_y prior to packets being exchanged directly between IP_x and IP_y.

3.1 Indirect estimation

Indirect estimation involves a group of nodes probing each other to establish baseline RTT knowledge, and a mechanism for inferring the RTT between IP_x and IP_y using information about other active nodes near IP_x and IP_y respectively. Examples include IDMaps [Francis et al. 2001], utilising regular active latency measurements made by *Tracers* located around the Internet, and King [Gummadi et al. 2002], using recursive queries through the domain name system (DNS) infrastructure to infer latency between any two IP_x and IP_y. However, this is suboptimal for FPS server discovery. In order to initially rank all IP_y by their RTTs, an IP_x would send at least one query packet to someone else for each IP_y whose RTT it wants to infer. The client might as well probe each IP_y directly (i.e. traditional FPS server discovery). Refinements such as Meridian [Wong et al. 2005] may directly answer questions such as "what nodes are within X ms of target T", but they require a cooperating infrastructure of Meridian nodes who actively probe each other and their targets, and forward or resolve queries on behalf of a querier, IP_x.

3.2 Embedded Network Coordinates

Global Network Positioning (GNP [Ng and Zhang 2001; 2002]) first introduced the idea of modeling the Internet as a virtual geometric space. Hosts are assigned
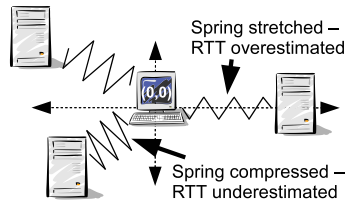
Fig. 3. Vivaldi: Differences between predicted and measured RTTs compresses or stretches 'springs', pushing the node to new coordinates.
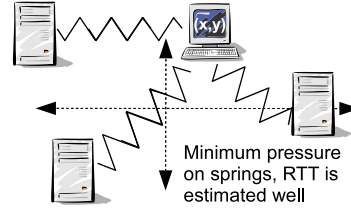
Fig. 4. Vivaldi: After many iterations, nodes are pushed to positions with minimum discrepancy between 'distance' and measured RTTs.

network coordinates such that the 'distance' between two hosts in an N-dimensional space is roughly proportional to the real RTT between them (this is known as *embedding*). If IP_x and IP_y learn each other's coordinates by indirect means, they can calculate the likely RTT between themselves prior to direct communication.

3.2.1 *Landmarks.* GNP utilised *Landmarks* – a set of cooperating hosts who actively measure RTTs between themselves. These RTTs are shared with a central node, which calculates coordinates for each Landmark by minimising the overall error between inter-Landmark distances and measured RTTs. Landmarks become anchor points in the coordinate space – other hosts calculate their own coordinates after measuring the RTTs to a subset of Landmarks whose coordinates are already known. A challenge for GNP is the need for computationally-intensive, centralised (re)calculation of Landmark coordinates whenever the full-mesh matrix of inter-Landmark RTTs changes (either due to the RTT measurements fluctuating over time, or Landmarks being turned on or off).

3.2.2 *Distributed incremental calculation.* Vivaldi [Dabek et al. 2004] eliminated Landmarks and introduced a decentralised system of incremental coordinate calculation that piggy-backed on pre-existing packet exchanges between participating nodes (such as occur between peers in a peer-to-peer file sharing system).

Vivaldi models nodes as masses connected by springs to other nodes. The rest length of each spring is the measured RTT between nodes. Figure 3 shows each spring under tension (or compression) if the rest length differs from the distance between each nodes' current coordinates. Node X measures the RTT to, and learns current coordinates of, node Y each time it communicates with node Y for some other reason. Node X then incrementally adjusts its own coordinates to minimise the potential energy in the spring notionally connecting X and Y. Figure 4 shows a node converged, after many communication exchanges, on a set of coordinates that are useful for RTT prediction. (Node X likewise regularly shares its coordinates with its peers so they may update their own coordinates.)

Vivaldi distributes the computational load of coordinate (re)calculation across both time and space. Nodes need not be a neighbor to every other node, and nodes update their coordinates incrementally and asynchronously (relative to their peers) as RTT measurements are made or neighboring nodes are turned on or off. Related techniques have also emerged that differ primarily in the details of how they position nodes in coordinate space.

3.2.3 *An imperfect predictor.* Network coordinates can be an imperfect predictor of both absolute distance and relative rank order [Lua et al. 2005; Ledlie et al. 2006]. One underlying assumption is that network paths adhere to the triangle inequality, which says distance vectors A and B between three nodes satisfy $|A| - |B| \leq |A + B| \leq |A| + |B|$. However, this may not always hold for paths between arbitrary internet hosts. RTT measurements vary over time as network conditions change, so coordinates calculated at time $t = T_0$ may not estimate RTT present at time $t > T_0$. One must use enough dimensions to accurately capture the complexities of Internet topology, yet not use more dimensions than required for sufficiently accurate RTT prediction. For example, [Dabek et al. 2004] showed 5D coordinates provide better accuracy than 2D or 3D, albeit with increasing computational complexity and diminishing improvement per extra dimension.

3.3   Using network coordinates to augment FPS Server Discovery

Despite the imperfections, network coordinates have shown promise when building multicast trees for group communication systems [Vik et al. 2009] and have excellent potential to augment traditional FPS server discovery.

If game servers and clients have network coordinates, we may estimate the relative distances between a client and all game servers *before* the client begins probing for game-state details. By subsequently probing from closest to furthest, clients can expect to learn the game state details of the closest N game servers within $(N + \delta)$ probes (for some small integer δ). This is a major improvement over the traditional approach of probing every game server in no particular order.

It is more important for coordinates to establish usable *relative* distances than accurate absolute distances. For example, game servers {X,Y,Z} would be probed in the same order whether their coordinates place them {30,70,90}ms or {35,50,110}ms from the client. Furthermore, the relative rankings need not be perfect. As noted in Section 2, a player has no a priori expectations of which game servers they will see first. At a plausible 100+ probes/second, modest misordering will be quickly hidden as the client dynamically updates its on-screen list of game servers ranked by the actual RTTs measured when each game server is probed.

4.   REED – NETWORK COORDINATES FOR FASTER SERVER DISCOVERY

Here we describe the REED architecture and illustrate REED as a variant of Steam's existing CS:S server discovery process in a virtual 4D coordinate space.

A REED master server (RMS) borrows key ideas from Vivaldi to establish and track the coordinates of registered game servers. REED clients borrow from GNP to establish their own coordinates by using selected game servers as Landmarks, then rely on the RMS to rank game servers by distance from the client when queried. REED clients may stop probing when sufficient 'close' game servers have been probed, or a player-specified RTT threshold is reached.

4.1   Centralised, incremental calculation of game server coordinates

REED performs centralised, incremental calculations of game server coordinates distributed over time. We borrow from Vivaldi the idea of incrementally nudging a node closer to its real coordinates over successive RTT measurements, using a model of interconnecting springs whose tensions are based on RTTs measured between

---

Algorithm 2 REED game server (re)registration with RTT sampling

---

(1) Game server (G) sends short UDP 'join' packet to REED master server (RMS)

(2) RMS returns a single UDP packet containing:
    (a) A unique challenge code,
    (b) $T_{registration}$ (number of seconds until re-registration is next required), and
    (c) A list $S_{probe} = \{S_1, S_2, ... S_N\}$, where S_x is the 6-byte <IP address:port> pair of another game server

(3) G issues normal server discovery probes to each member of S_{probe}, creating a list $R_{probe} = \{R_1, R_2, ... R_N\}$, where R_x is the RTT measured to S_x

(4) G returns a single 'challenge-response' UDP packet to the RMS, containing the unique challenge code, local server details and the list R_{probe}

---

neighbors. We differ from Vivaldi in that the RMS both controls when neighbors (game servers) probe each other, and centrally updates each neighbor's coordinates using the measured RTTs. Beneficially our proposal minimises the cooperation and trust required *between* independently operated game servers.

Algorithm 2 extends the CS:S approach in Section 2.1.1 to now collect RTT samples as game servers regularly re-register. The RMS requests each re-registering game server to issue normal server discovery probes (as a client would) to a small set of 'neighbor' game servers (the S_{probe} list) and then return these measured RTTs (the R_{probe} list). REED assumes valid RTTs lie in the range $0 \leq R_x \leq 998ms$, with $R_x = 999$ indicating server S_x can be ignored (it did not answer within $998ms$). Every member of S_{probe} thus has a matching member in R_{probe}. The RMS minimises global synchronisation of registrations by modulating $T_{registration}$ (time to next re-registration) as game servers re-register. ($T_{registration} = 300$ and $S_{probe} = \{\}$ would mimic conventional CS:S, where game servers exchange three packets with the master server every five minutes.)

A game server's coordinates are initialised to (0,0,0,0) when it first registers. Each time it re-registers, the RMS selects a set of target neighbors to probe (S_{probe}), and uses the returned RTT measurements to drive incremental (Vivaldi-*like*) updates to game server coordinate stored in the RMS. Over time all game server coordinates converge to a useful set of values, and RMS memory use scales with the total number of registered game servers. Section 7.2.1 discusses possible S_{probe} selection strategies trading timely convergence against number of probes emitted per re-registration.

REED treats RTT measurements as bidirectional – when game server G_a probes game server G_b, the RMS treats this as though G_b also probed G_a. Multiple game servers may use different ports to share an IP address – the RMS treats these as being the same node when constructing S_{probe} and calculating coordinates.

4.2 Calibration – using landmarks to position a client in coordinate space

REED clients begin server discovery by using Algorithm 3 to position themselves relative to a set of landmark game servers with known coordinates. A *getCalib-Servers* message requests N_{calib} game servers and the RMS selects and returns these landmarks (the $S_{calibprobe}$ list) in a *getCalibServersResponse* reply. Direct probing then establishes the RTT to each server, and the client locally calculates its own coordinates (C_w, C_x, C_y, C_z) such that they minimise the overall error between

---

Algorithm 3 Calibration: Positioning a game client in coordinate space

---

(1) Game client (C) sends a *getCalibServers* query to the RMS, indicating the number of game servers C wishes to probe (N_{calib})

(2) RMS sends a *getCalibServersResponse* packet to C, containing a list $S_{calibprobe} = \{S_1, A_1, S_2, A_2, ... S_N, A_N\}$, where $N \leq N_{calib}$ and A_x represents the (w, x, y, z) coordinates of a *landmark* game server whose 6-byte <IP address:port> pair is S_x

(3) C measures RTT to each member of $S_{calibprobe}$ using normal server discovery probes

(4) Using the measured RTTs, C calculates its probable location (C_w, C_x, C_y, C_z) in coordinate space relative to the coordinates (A_x) of each member of $S_{calibprobe}$

Algorithm 4 Probing in order of increasing distance from client

---

(1) Game client (C) sends a *getOrderedServers* query to the RMS containing:
 — (C_w, C_x, C_y, C_z), the client's present location
 — RTT_{limit}, only game servers closer than this RTT in ms ($\geq 999ms$ for "all servers")
 — S_{last}, the last 6-byte <IP address:port> pair returned in the immediately preceding *getOrderedServersResponse* (or <0.0.0.0:0> if this is the first query)

(2) C receives a *getOrderedServersResponse* containing $S_{probe} = \{S_1, S_2, ... S_N\}$, where
 — Members of S_{probe} are pre-sorted in order of increasing euclidean distance from C
 — S_x is the 6-byte <IP address:port> pair of an individual game server
 — N is limited by packet size ($N \leq 230$ using Steam's packets, Section 6.4)

(3) C issues regular server discovery probes to each member of S_{probe} in turn

(4) C repeats steps 1, 2 and 3 until the RMS has no more game servers to return (indicated by the last member of S_{probe} being <0.0.0.0:0>) or the client terminates early (Sections 4.3.2 or 4.3.3)

---

measured RTTs and euclidean distances to the coordinates (A_x) of each landmark (S_x). Section 7.2.2 discusses strategies for selecting members of $S_{calibprobe}$.

For N-dimensional space we need $N_{calib} \geq (N+1)$ landmarks. (In Section 5.2 we use $N_{calib} = 14$ and find diminishing benefits to $N_{calib} \geq 12$ using 4D coordinates.)

4.3 Ordered Probing and Early Termination (Auto-stop)

After establishing its own coordinates, a REED client performs ordered probing of game servers using Algorithm 4 (based on Algorithm 1). We describe four possible techniques for automatic early termination of probing (*auto-stop*) once a player has enough information from which to make their selection (evaluated in Section 5). We assume the client lets a player specify their maximum tolerable RTT (RTT_{stop}) and/or how many of the closest game servers they wish to chose from (N_{close}).

4.3.1 *Probing in order of increasing distance from client.* Upon receiving a client's initial *getOrderedServers* query ($S_{last} = $ <0.0.0.0:0>) the RMS ranks all registered game servers in order of ascending distance from the client's declared location, (C_w, C_x, C_y, C_z), up to a nominated RTT_{limit} (for $RTT_{limit} \leq 998ms$, or $RTT_{limit} = 999ms$ to return all game servers). The client then uses repeated exchanges of *getOrderedServers* and *getOrderedServersResponse* to retrieve and probe this ordered list of game servers.

By using the RMS to rank game servers the client can begin probing once the

---

Algorithm 5 Early termination when RTTs exceed player's threshold (RTT_{stop})

---

(1) $W_{autostop}$ is the sampling window size (e.g., $W_{autostop} = 100$)

(2) Wait for at least $W_{autostop}$ servers to be probed

(3) Terminate Algorithm 4's probing when $RTT_{bottom} \geq RTT_{stop}$, where RTT_{bottom} is the 2nd percentile of the last $W_{autostop}$ RTT samples

---

Algorithm 6 'Intelligent' probing of N_{close} game servers

---

(1) $W_{autostop}$ is the sampling window size (e.g., $W_{autostop} = 100$)

(2) N_{close} is the number of suitable servers to return

(3) Terminate Algorithm 4's probing when $B \geq N_{close}$, where:
   —B is the number of probed game servers with RTTs below the current value of RTT_{bottom}, and RTT_{bottom} is the 2nd percentile of the last $W_{autostop}$ RTT samples

---

first *getOrderedServersResponse* message arrives. This also avoid sending every game server's coordinates to the client, which maximises the number of servers we can fit in each *getOrderedServersResponse* message packet.

4.3.2   *Terminate after probing all servers under RTT_{stop}.* Algorithm 4 relies on the RMS calculating reasonably correct *absolute* distances. In practice a client using Algorithm 4 alone should set $RTT_{limit} = (RTT_{stop} + \delta_{rtt})$. The positive offset ($\delta_{rtt}$) helps minimise the number game servers excluded because their RTTs are estimated to be over, even when actually under, a player's nominated RTT_{stop}.

An alternative is to set $RTT_{limit} = 999ms$ and implement Algorithm 5 – a client-side auto-stop approach from [Armitage 2008b] that presumes the RMS gets *relative* rankings reasonably correct so measured RTTs generally trend upwards. Algorithm 5 terminates Algorithm 4 when the 2nd percentile of recent RTTs exceeds RTT_{stop} (to minimise premature auto-stop due to outliers whose RTTs diverge noticeably from the distance implied by their coordinates).

Unfortunately, neither scheme helps clients who are close to the majority of game servers. For example, English CS:S clients (Section 2.2.3) would probe over 90% of all CS:S servers if we used Algorithm 5 with a quite realistic $RTT_{stop} \leq 150-200ms$.

4.3.3   *Terminate after probing N_{close} suitable servers.* A player who triggers server discovery seeks a practical selection of N_{close} nearby game servers, not simply 'the closest' game server nor the thousands whose RTTs may fall below RTT_{stop}.

We propose two extensions to Algorithm 4. Our 'First' method is to simply query the first N_{close} game servers in the order they are returned by the RMS (regardless of whether each probed server's RTT is consistent with its relative rank).

Our 'Intelligent' method adapts to any divergence between measured RTTs and the master server's predicted distances. Algorithm 6 terminates Algorithm 4 once we have probed N_{close} game servers whose RTTs are under a (slowly increasing) lower threshold, RTT_{bottom}. By presenting the player only those game servers whose RTTs are under RTT_{bottom}, we remove from consideration any outliers whose RTTs noticeably diverge from the distance implied by their coordinates.

(A trivial extension to both methods would be to probe N_{close} game servers that also meet additional criteria, such as "has X players" or "uses map Y".)

5.   IMPACT OF REED ON CLIENT PERFORMANCE

Our core contribution is to show the extent to which REED can benefit clients under typical conditions. An RMS would usually operate for months at a time, so for our typical case we treat the RMS as having already established reasonably stable coordinates for registered game servers. Hypothetically assuming Steam was upgraded to use REED, we illustrate REED's positive impact on a CS:S client's server discovery process for clients located in three different parts of the planet, and evaluate the impact of Section 4.3's different auto-stop techniques.

5.1   Methodology – Emulating a deployed REED system

Upgrading all CS:S servers and clients with REED was impractical. Instead we constructed two matrices of live RTT measurements involving active CS:S game servers in mid-2009, and used this to emulate both the calculation of game server coordinates by an RMS and a client's calibration and ordered probing sequence.

   Network coordinates works best when the number of dimensions, and shape of the virtual space, capture the vagaries of internet routing. We evaluated regular 2D, 3D, 4D and 10D coordinates, and the 2D+H ("height") coordinate scheme proposed for Vivaldi [Dabek et al. 2004].

   5.1.1   *Emulating the REED master server's positioning of game servers.* First, we configured a real CS:S game server on each one of 15 PlanetLab (PL) [PlanetLab 2008] nodes around the world (five each from Europe, North America and Asia). Each of these PL nodes then probed all ($\sim$36 000) registered CS:S game servers multiple times (including those on the other PL nodes) to build up a matrix of RTT samples (using Qstat, a command-line game server browser, `http://www.qstat.org`) . Each PL node probes the CS:S game servers on every other PL node four times back to back, then probes all other CS:S game servers over the next four to five minutes. We repeated this process five times, once every 25 minutes.

   Using the minimum RTTs captured across multiple measurements (the ones least-influenced by transient network congestion) we built a 15x15 matrix of RTTs between the PL nodes, and a 15xN matrix of RTTs between each PL node and the $\sim$36 000 CS:S game servers. We calculated 2D, 3D, 4D, 10D and 2D+H coordinates for all CS:S game servers in two steps. First, we used the 15x15 matrix of RTT measurements to calculate the positions of our 15 PL nodes relative to each other (in each coordinate space). These 15 PL nodes were then used as landmarks to calculate the coordinates of every other active CS:S game server (using the RTT measurements in our 15xN RTT matrix).

   5.1.2   *Emulating a client's calibration and ordered probing steps.* We evaluated the impact of REED from the perspective of five clients in Europe, America and Asia respectively. To do so, each of the 15 landmark locations were, in turn, treated as a client location that is presumed to have probed the remaining 14 landmark nodes to establish its coordinates. (In other words, $S_{calibprobe}$ from Algorithm 3 always contains the *other* 14 landmark PL nodes, and probing is emulated by reading RTTs from the 15x15 RTT matrix created in Section 5.1.1.) For each client we then sorted the $\sim$36 000 CS:S game servers by ascending 'distance' from the client (using coordinates calculated in Section 5.1.1 for the CS:S game servers).
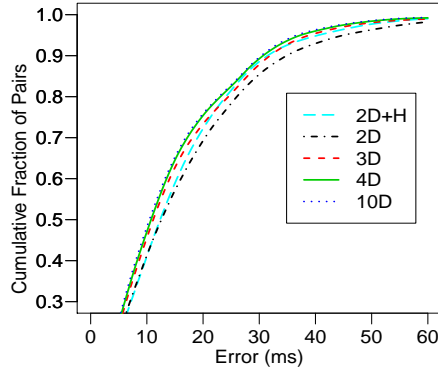
Fig. 5. Distance estimation error between pairs of PL nodes (landmarks) and CS:S game servers when positioned as in Section 5.1.1 with 2, 3, 4 and 10 dimensions or 2D+H coordinates
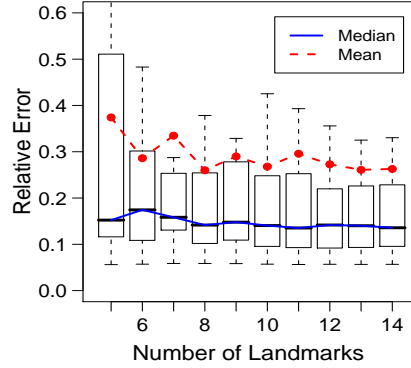
Fig. 6. Relative error between estimated and measured RTT in 4D space vs number of landmarks (across all clients to all 36K CS:S game servers)

Each client's ordered probing was then emulated by looking up (in the 15xN RTT matrix) previously measured RTTs between the client and each game server.

5.1.3 *Choice of coordinate space and number of landmarks.* Coordinate systems trade complexity against estimation accuracy. Figure 5 compares the distribution of absolute error between estimated and measured RTTs for every pair of landmark (PL) nodes and 36K CS:S game servers when using 2D, 3D, 4D or 10D euclidean coordinates or the 2D+H scheme. 2D performs worst, 2D+H is similar to 3D coordinates[3], and 4D is better again and indistinguishable from 10D. Thus we chose 4D coordinates for our detailed analysis of ordered probing and auto-stop.

Embedding a client in an N-dimensional coordinate space requires probing at least $N + 1$ landmarks, but not too many more landmarks than are required for a reasonable estimate of the client's coordinates. Using 4D coordinates, Figure 6 shows the distribution of error between estimated and measured RTTs measured from all 15 clients to all 36K CS:S game servers versus number of landmarks use to establish each client's coordinates. Relative error stabilises for ≥ 12 landmarks, so our use of 14 landmarks in Section 5.1.2 and subsequent analysis is reasonable.

5.2 Results from a REED client's perspective

Here we show the degree to which REED clients probe game servers in ascending order of actual RTT, and consider each of Section 4.3's auto-stop techniques.

5.2.1 *Ordered probing results in generally ascending RTT.* Figures 7, 8 and 9 show the initial 14 calibration probes and subsequent ordered probing experienced by REED clients in Europe, America and Asia respectively. Compared to Figure 2, REED clients see generally ascending measured RTTs and many low-RTT game servers within the first few seconds of probing. This strongly suggests that REED clients can, and should, implement one of Section 4.3's auto-stop techniques.

---

[3]Vivaldi intended the H term to capture the latencies of access links, assuming that no two nodes share an access link. This assumption is often violated by groups of FPS game servers.
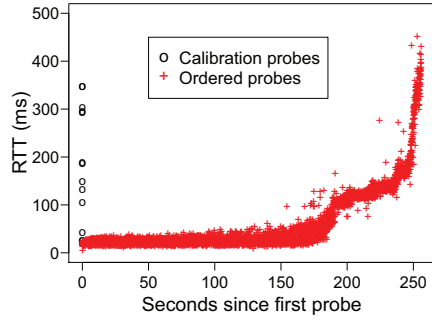
Fig. 7. Probed RTT versus time for European CS:S client using REED at 140 probes/second
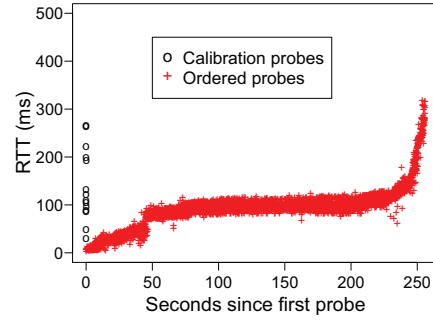

Fig. 8. Probed RTT versus time for American CS:S client using REED at 140 probes/second
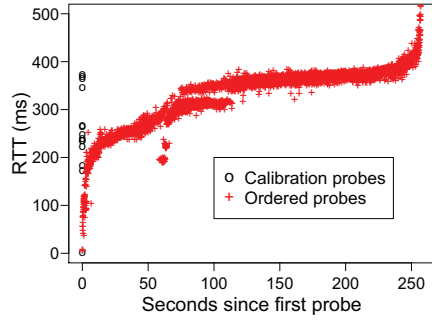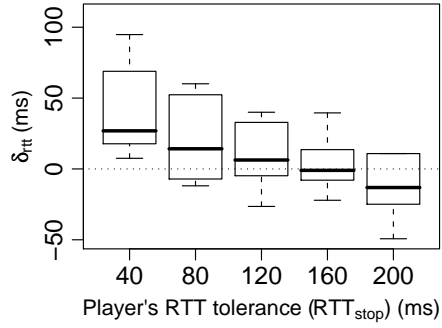

Fig. 9. Probed RTT versus time for an Asian CS:S client using REED at 140 probes/second

Fig. 10. The δ_{rtt} required by Algorithm 4 to cover 98% of game servers under RTT_{stop}

RTT_{stop}(ms)	40	80	120	160
Europe	29	25	13	5
America	88	78	12	3
Asia	99	99	98	97

Table I. Percentage reduction in probe time and network traffic for clients in Europe, America and Asia when using Algorithm 5 for auto-stop

RTT_{stop}(ms)	40	80	120	160
Europe	0.05	0.03	0.04	0.02
America	1.03	0.12	0.03	0.03
Asia	0.00	0.50	1.18	1.66

Table II. Percentage of servers under RTT_{stop} that are missed by clients in Europe, America and Asia when using Algorithm 5 for auto-stop

5.2.2   *Auto-stop after exceeding RTT_{stop}.* Both of Section 4.3.3's auto-stop techniques are poor for clients who see thousands of game servers under RTT_{stop}.

Imagine we wished to use Algorithm 4 alone (with $RTT_{limit} = (RTT_{stop} + \delta_{rtt})$) and aimed (for somewhat arbitrary reasons) to discover at least 98% of all game servers under RTT_{stop}. Figure 10 shows the spread of δ_{rtt} required to achieve this goal for a range of RTT_{stop}. For $RTT_{stop} = 40ms$ we need RTT_{limit} of $60 - 90ms$. The required δ_{rtt} also varies significantly with RTT_{stop}, so deploying clients using Algorithm 4 and a fixed δ_{rtt} is impractical.

Table I reveals the reduction in probe time and traffic experienced by different clients when using Algorithm 5 and various RTT_{stop}. Asian clients see a substantial

40 Servers with First auto-stop			
	Eu	Am	As	All
Mean RTT	17.6	10.7	47.9	25.4
Median RTT	19.0	11.5	39.1	23.2
Reduction	99%	99%	99%	99%

120 Servers with First auto-stop			
	Eu	Am	As	All
Mean RTT	18.6	12.5	76.2	35.8
Median RTT	19.1	12.6	72.2	34.6
Reduction	99%	99%	99%	99%

Table III. *First* auto-stop for 40 servers – 54 probes in 0.39 seconds

Table IV. *First* auto-stop for 120 servers– 134 probes in 0.96 seconds

40 Servers with Intelligent auto-stop			
	Eu	Am	As	All
Mean RTT	11.0	6.8	23.3	13.7
Median RTT	11.8	7.8	28.2	15.9
Mean Probes	771	646	184	534
Mean Time	5.5s	4.6s	1.3s	3.8s
Reduction	97%	97%	99%	98%

120 Servers with Intelligent auto-stop			
	Eu	Am	As	All
Mean RTT	13.1	9.1	63.2	28.5
Median RTT	13.2	9.7	66.5	29.8
Mean Probes	1835	837	438	1037
Mean Time	13.1s	6.0s	3.1s	7.4s
Reduction	93%	97%	98%	96%

Table V.   *Intelligent* auto-stop for 40 servers

Table VI. *Intelligent* auto-stop for 120 servers

benefit for all values of RTT_{stop}. However, the abundance of nearby game servers means European clients receive a modest 28% reduction at $RTT_{stop} = 40ms$ and American clients see little benefit unless $RTT_{stop} \leq 100ms$.

Table II shows the percentage of servers under a particular RTT_{stop} who would *not* have been probed by clients in each region when using Algorithm 5. The RMS gets relative server rankings reasonably correct, so clients in all three regions would probe until almost all game servers under each nominated RTT_{stop} are found. However, as previously noted, this is not a good outcome for clients where many thousands of game servers exist under the player's RTT tolerance.

5.2.3 *Auto-stop after probing N_{close} servers.* Much better results are achieved using Section 4.3.3's First and Intelligent methods for probing a sufficient number of the game servers estimated to be close by (N_{close}). We consider two scenarios: $N_{close} = 40$ and $N_{close} = 120$.

Tables III and IV show the impact of using the First auto-stop technique for European (Eu), American (Am) and Asian (As) clients sending 140 probes per second. The per-region means and medians are derived using separate measurements from all five clients in each region. The RTTs are derived from the game servers that are ultimately presented to the player. All regions need one second or less to identify 40 or 120 game servers with quite playably low RTTs.

Relative to First auto-stop, Tables V and VI show that Intelligent auto-stop identifies 40 or 120 game servers with generally lower mean and median RTTs, but requires roughly 5 to 30 times more probes. (As European clients see a significant number of low RTT game servers, their use of Intelligent auto-stop results in roughly four times as many probes as American clients and over six times as many probes as required by Asian clients.)

5.2.4 *Choice of auto-stop algorithm.* Both First and Intelligent auto-stop achieve a substantial reduction in network traffic and probing time relative to the ∼36 000 probes emitted by a regular CS:S client. Which approach to implement depends
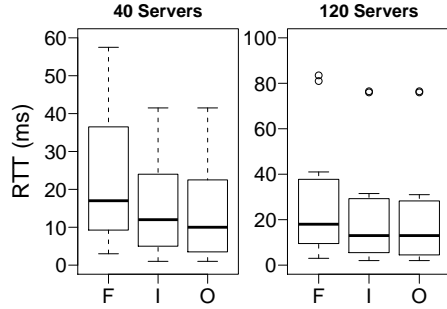
Fig. 11. Spread of game server RTTs seen by all 15 clients using <u>F</u>irst and <u>I</u>ntelligent auto-stop or <u>O</u>ptimal ($N_{close} = 40$ or $N_{close} = 120$)
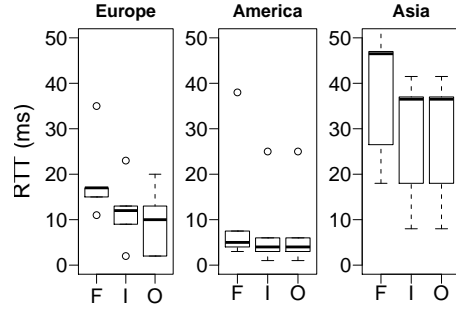
Fig. 12. Spread of game server RTTs seen by each region's clients using <u>F</u>irst and <u>I</u>ntelligent auto-stop or <u>O</u>ptimal ($N_{close} = 40$)

on whether it is more important to probe as quickly as possible or identify game servers whose RTTs are as low as possible.

Figure 11 shows the spread of game server RTTs obtained using First and Intelligent auto-stop versus the hypothetical 'Optimal' case (which assumes we select the closest N_{close} game servers, using perfect knowledge after all available game servers are probed). Whether we're identifying 40 or 120 playable game servers, Intelligent auto-stop selects a set of servers whose spread of RTTs is closer to Optimal. First auto-stop requires far fewer probes to select 40 or 120 game servers with a quite playable (albeit wider) spread of RTTs. Figure 12 shows that this relationship between First, Intelligent and Optimal RTT distributions also holds within each region. (To save space, a similar relationship for $N_{close} = 120$ is not shown.)

We recommend REED clients implement either First or Intelligent auto-stop with player-configurable N_{close}, allowing players themselves to control the trade-off between rapid or comprehensive server discovery.

6. DEPLOYMENT AND PERFORMANCE CONSIDERATIONS

Here we consider some of the performance issues relevant to realistic deployment of REED – how an RMS can handle hundreds of client requests per minute, the network load experienced by individual game servers, potential for third-party deployment of REED, and re-using Valve's existing control packet format.

6.1 Using GPUs for high-speed ranking of game servers

In early 2010 we observed that a single CS:S game server receives up to 350 *A2S_INFO Request* unique probes per minute from around the world. Handling the corresponding load of initial client queries to an RMS under REED would require an RMS to re-rank over 36 000 game servers up to 350 times per minute. Ranking points by relative distance is trivial for the graphics processing units (GPUs) in modern, low-cost consumer graphics cards. For example, an Nvidia 8800GT GPU can do roughly 24 000 new rankings per minute on 50 000 random points in 4D coordinate space using the Thrust CUDA library [Hoberock and Bell 2010].

A further simplification is to eliminate the $\sqrt{X}$ step to calculate absolute distances, and rank by $(distance)^2$ instead – this makes no difference to actual ordering

of game servers returned using Algorithm 4. Thus an RMS may easily and cheaply rank games servers with raw GPU power rather than rely on elegant algorithms.

6.2   Network traffic load on REED game servers

It is important to place REED's additional game server probing into context. Current CS:S game servers exchange three packets every five minutes with the master server during re-registration, or 36 packets/hour. In early 2010 CS:S game servers answered an average of 6120 server discovery probes/hour from clients worldwide over 24hr periods, thus exchanging 12240 packets/hour whether the game server had players or not. A game server with players experiences a further ~ 60 packets per *second* (or 216 000 packets/hour) bi-directionally per connected client.

In N-dimensional space game servers must be 'linked' by RTT measurement to at least $(N+1)$ other game servers. A REED game server being probed by X other game servers, and asked to probe Y targets, will see $3 + 2(X + Y)$ packets every 5 minutes due to re-registration. Consider a REED CS:S game server probed by 5 game servers, and probing 10 other game servers, every 5 minutes (X = 5, Y = 10). This game server's REED re-registration traffic is 396 packets/hour, or $\sim 3\%$ of the traffic from regular client probing. A whole day of REED re-registration traffic will be exceeded by a mere 3 minutes of game play by one player. Given that game servers register to attract players, REED re-registration traffic will range from being a trivial, to practically irrelevant, fraction of a game server's overall network load. Furthermore, re-registration traffic for most game servers will be offset by a reduction in regular probing by REED clients (how much will depend on relative distribution of each particular game type's game servers and player populations).

6.3   Third-party deployment of REED

Using just 15 landmarks regularly probing 36K game servers, Section 5's results suggest REED might also be utilised by third-party server browsers (such as GameSpyArcade, `http://www.gamespyarcade.com`). Deploy 15 or more well-connected private landmark nodes around the world to probe each other and all active game servers every 5 minutes (analogous to Section 5.1's use of PlanetLab nodes). Let a third-party 'RMS' regularly collect these RTT samples from each landmark, (re)calculate coordinates for all game servers and answer queries from the third-party browser's own REED client. With 15 landmarks, game servers would experience 180 additional probes per hour but emit no new probes during re-registration.

6.4   Implementing Algorithm 2 using Steam packet formats

If we re-use Steam's existing packet format an RMS may request up to 230 RTT samples each time a game server re-registers (Algorithm 2). Steam's 1400 byte UDP payload limit [Valve Corporation 2009c] allows $N \leq 230$ entries in a query packet's S_{probe} list if it is encoded as a sequence of 6-byte binary values terminated by <0.0.0.0:0> (allowing 10 bytes the RMS's 4-byte challenge code and $T_{registration}$). Steam's challenge-response packet utilises backslash-delimited clear text. Treating each R_x as a three digit string in the range $000 \leq R_x \leq 999$, R_{probe} may be encoded as a string of $3N$ digits taking up to 690 bytes. Current CS:S challenge-response payloads are typically less than 256 bytes, so there is plenty of room to add up to 690 bytes of R_{probe} within a 1400 byte UDP payload.

7.   LIMITATIONS AND FUTURE WORK

Players are fully aware of the uncertainties in server discovery – servers come and go, network conditions change, so players do not expect to always find the same servers in the same order every time. We've evaluated REED from the perspective of reducing client probing time and network traffic, assuming an RMS that's been operating for months and established reasonable coordinates for its long-term registered game servers. This leaves a number of open questions for future work.

7.1   Methodological limitations

We used well-connected PlanetLab nodes as both landmarks and 'fake clients'. As REED's estimation of game server coordinates primarily depends on probing *between* game-servers (which are typically also well-connected) we believe our methodology reasonably demonstrates REED's potential benefits. Where probing times are mentioned, we have assumed the time taken to query the RMS is negligible and that client probes are emitted in a pipelined manner at 140 probes/second.

7.2   Challenges for the REED master server

7.2.1   *Chosing S_{probe} for timely convergence.* Timely convergence (in response to network variations and game servers starting up or shutting down) must be balanced against the number of probes each game server emits per re-registration. A large S_{probe} set provides quick convergence for recently registered game servers, while long-lived game servers may be sufficiently served by small S_{probe} sets. At minimum, in N-dimensional coordinate space every game server must be a direct neighbor of ('linked' by RTT measurement to) at least $(N+1)$ other game servers, and a path must exists (connecting via one or more neighbors) between any two game servers. Consequently there remains fruitful future work studying the impact of different RMS strategies for populating S_{probe} and varying $T_{registration}$ over time. Possibilities include ensuring each S_{probe} contains a mix of game servers from different countries (geo-location based on IP addresses), or including more active rather than inactive game servers, or building specific types of interconnected partial-meshes or trees of links in coordinate space, and so on.

7.2.2   *Chosing $S_{calibprobe}$ landmarks.* Section 5.1.2's client calibration utilised landmarks from three major geographical regions, providing a diverse set of reference points. However, the RMS might instead select landmarks scattered uniformly around the coordinate space's origin, or use geo-location to identify landmarks who (geographically) surround the querying client. In addition, the RMS should change landmarks over time so calibration probe traffic is not focused on any one set of game servers. Future work could characterise the performance implications of different strategies for choosing Algorithm 3's $S_{calibprobe}$ set for each querying client.

7.2.3   *Impact of anomalous RTT samples.* By varying S_{probe} over time, REED minimises the incentive for any given game server to disruptively influence RTT samples. For example, imagine that game server X induces induces high apparent RTT between itself and game server Y (by lying about the RTT it measures to Y, and/or delaying its replies to Y's own probes). Any negative impact on server Y's coordinates will be diluted as the RMS continues to solicit and receive un-tainted

estimates of RTT between Y and other game servers.

If temporary, localised network congestion causes a game server to return inflated RTTs in their R_{probe} lists the impact will similarly be diluted by time (as more samples arrive) and space (as the RMS utilises reports from multiple neighbors to adjust the coordinates of any given game server).

Future work should explore the degree to which the RMS can detect, and mitigate against, anomalous RTT samples (whether deliberate or accidental), and provide incentives for independent game server operators to act in 'good faith'.

8. CONCLUSION

REED is a novel application of previous work on network coordinates to the problem of game server discovery where publisher-hosted master servers track independent, enthusiast-hosted game servers. A REED master server (RMS) directs REED game servers to regularly sample the RTTs between themselves, then uses these samples to embed game servers into a virtual coordinate space. In effect, REED performs centralised, incremental calculations of game server coordinates distributed over time. REED clients establish their own coordinates by using selected game servers as landmarks (without needing local geo-location knowledge). The RMS ranks and returns game servers by distance from clients, so clients may probe game servers by ascending order of likely RTT regardless of the client's location. Using data on over 36K Counter Strike:Source (CS:S) game servers, collected by PlanetLab landmark nodes, we show REED reduces a CS:S client's probe time and network traffic levels to as little as 1% of conventional server discovery. While conventional CS:S clients generate tens of thousands of UDP flows over multiple minutes, REED clients discover 40-100 playable, low-RTT game servers in $1 - 13$ seconds while using $\sim 54 - 1800$ UDP flows – expediting a player's ability to select and join a game server, and curtailing the creation of unnecessary UDP/IP flow-state in NAT-enabled gateways or wireless access points. REED can improve server discovery of any game whose diversely located clients and servers are often separated by more than the RTT typically accepted for competitive online game-play.

9. ACKNOWLEDGEMENT

REFERENCES

AGARWAL, S. AND LORCH, J. R. 2009. Matchmaking for online games and other latency-sensitive p2p systems. In *Proc. of ACM SIGCOMM 2009 conference on Data communication*. ACM, New York, NY, USA, 315–326.

ARMITAGE, G. 2008a. Client-side Adaptive Search Optimisation for Online Game Server Discovery. In *Proc. of IFIP/TC6 NETWORKING 2008*. Singapore.

ARMITAGE, G. 2008b. Optimising Online FPS Game Server Discovery through Clustering Servers by Origin Autonomous System. In *Proc. of ACM NOSSDAV 2008*. Braunschweig, Germany.

ARMITAGE, G. September 2003. An Experimental Estimation of Latency Sensitivity in Multiplayer Quake3. In *Proc. of 11th IEEE International Conference on Networks*. Sydney, Australia.

ARMITAGE, G., JAVIER, C., AND ZANDER, S. December 2006. Topological optimisation for online first person shooter game server discovery. In *Proc. of Australian Telecommunications and Network Application Conference*. Sydney, Australia.

BEIGBEDER, T., COUGHLAN, R., LUSHER, C., PLUNKETT, J., AGU, E., AND CLAYPOOL, M. 2004. The effects of loss and latency on user performance in Unreal Tournament 2003. In *Proc. of 3rd workshop on Network and system support for games*. ACM, New York, NY, USA, 144–151.

CHAMBERS, C., FENG, W.-C., W.-C., F., AND SAHA, D. 2003. A geographic, redirection service for on-line games. In *ACM Multimedia 2003 (short paper)*.

CLAYPOOL, M. 2008. Network characteristics for server selection in online games. In *ACM/SPIE Multimedia Computing and Networking (MMCN)*.

DABEK, F., COX, R., KAASHOEK, F., AND MORRIS, R. 2004. Vivaldi: a decentralized network coordinate system. In *Proc. of ACM SIGCOMM 2004 conference on Data communication*. ACM, New York, NY, USA, 15–26.

FENG, W.-C. AND FENG, W.-C. 2003. On the geographic distribution of on-line game servers and players. In *Proc. of the 2nd workshop on Network and system support for games*. ACM Press, New York, NY, USA.

FRANCIS, P., JAMIN, S., JIN, C., JIN, Y., RAZ, D., SHAVITT, Y., AND ZHANG, L. 2001. Idmaps: a global internet host distance estimation service. *IEEE/ACM Transactions on Networking 9,* 5 (Oct), 525–540.

GUMMADI, K. P., SAROIU, S., AND GRIBBLE, S. D. 2002. King: estimating latency between arbitrary internet end hosts. In *Proc. of the 2nd ACM SIGCOMM Workshop on Internet measurment*. ACM, New York, NY, USA, 5–18.

HENDERSON, T. 2002. Observations on game server discovery mechanisms. In *Proc. of the 1st workshop on Network and system support for games*. ACM, New York, NY, USA, 47–52.

HOBEROCK, J. AND BELL, N. 2010. Thrust c++ template library for CUDA. `http://code.google.com/p/thrust/`.

LEDLIE, J., PIETZUCH, P., AND SELTZER, M. 2006. Stable and accurate network coordinates. In *Proc. of the 26th IEEE International Conference on Distributed Computing Systems*. IEEE Computer Society, Washington, DC, USA, 74.

LEE, Y., AGARWAL, S., BUTCHER, C., AND PADHYE, J. 2008. Measurement and Estimation of Network QoS Among Peer Xbox 360 Game Players. In *Proc. 9th Passive and Active Network Measurement Conference (PAM 2008)*. Springer Berlin / Heidelberg, 41–50.

LUA, E. K., GRIFFIN, T., PIAS, M., ZHENG, H., AND CROWCROFT, J. 2005. On the accuracy of embeddings for internet coordinate systems. In *Proc. of the 5th ACM SIGCOMM conference on Internet Measurement*. USENIX Association, Berkeley, CA, USA, 11–11.

NG, T. AND ZHANG, H. 2002. Predicting internet network distance with coordinates-based approaches. In *Proc. of IEEE INFOCOM 2002*. Vol. 1. 170–179.

NG, T. S. E. AND ZHANG, H. 2001. Towards global network positioning. In *Proc. of the 1st ACM SIGCOMM Workshop on Internet Measurement*. ACM, New York, NY, USA, 25–29.

PLANETLAB. 2008. Planetlab - an open platform for developing, deploying, and accessing planetary-scale services. `https://www.planet-lab.org/`.

VALVE CORPORATION. 2009a. Counterstrike: Source. `http://counter-strike.net/`.

VALVE CORPORATION. 2009b. Master server query protocol. `http://developer.valvesoftware.com/wiki/Master_Server_Query_Protocol`.

VALVE CORPORATION. 2009c. Server queries. `http://developer.valvesoftware.com/wiki/Server_Queries`.

VALVE CORPORATION. 2009d. Welcome to Steam. `http://www.steampowered.com/`.

VIK, K.-H., GRIWODZ, C., AND HALVORSEN, P. 2009. On the influence of latency estimation on dynamic group communication using overlays. R. Rejaie and K. D. Mayer-Patel, Eds. *Multimedia Computing and Networking 2009 7253,* 1, 725307.

WONG, B., SLIVKINS, A., AND SIRER, E. G. 2005. Meridian: a lightweight network location service without virtual coordinates. In *Proc. of ACM SIGCOMM 2005 conference on Data communication*. ACM, New York, NY, USA, 85–96.

ZANDER, S., KENNEDY, D., AND ARMITAGE, G. October 2005. Dissecting server-discovery traffic patterns generated by multiplayer first person shooter games. In *Proc. of 4th workshop on Network and system support for games*. New York, USA.