Distributed Server Replication in Large Scale Networks[*]

Bong-Jun Ko
Department of Electrical Engineering
Columbia University
New York, NY
kobj@ee.columbia.edu

Dan Rubenstein
Department of Electrical Engineering
Columbia University
New York, NY
danr@ee.columbia.edu

ABSTRACT

Quality of service for high-bandwidth or delay-sensitive applications in the Internet, such as streaming media and online games, can be significantly improved by replicating server content. We present a decentralized algorithm that allocates server resources to replicated servers in large-scale client-server networks to reduce network distance between each client and the nearby replicated server hosting the resources of interest to that client. Preliminary simulation results show that our algorithm converges quickly to an allocation that reduces the expected client-server distance by almost half compared to the distance when the assignment of replicated servers is done at random.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Distributed networks; C.2.4 [**Distributed Systems**]: Client/server

General Terms

Algorithms, Performance, Design

Keywords

Server replication, Distributed algorithm, Convergence

1. INTRODUCTION

The emergence of Internet applications that demand large bandwidth or small delay has motivated content (or service) providers to replicate contents or resources to provide better quality of service to their clients. For instance, streaming video, which requires relatively large bandwidth, is often distributed via Content Distribution Networks(CDN)[1]. CDNs replicate contents to multiple points in the Internet, serving users with the closest copy, thereby bypassing bottleneck points in the Internet. In networked online games[6, 9], which are traditionally supported via a client-server architecture, a game player's gaming experience is negatively affected by large propagation delay between server and clients. Game providers often utilize multiple game servers such that users can play the game at a nearby server to achieve a small delay.

Since these applications typically strive to serve massively large number of clients, application providers must deploy a large number of replicated servers to provide satisfactory service to all users, sometimes incurring excessive deployment and management costs. For this reason, application service providers often rely on outsourcing service from third-party server-network providers, who offer to their customers (i.e., content/application providers) services such as server hosting, placement and management, and end-user interfaces (e.g., user redirection, server directory). Rather than use dedicated servers for each content owner, these hosting service providers typically use a common pool of servers to host the needs of their customers, and assign available server resources dynamically to each content owner as a function of its demand.

In this paper, we focus on the problem of assigning replicated resources to servers in large-scale client-server networking environments, where we would like to place server replicas such that each client can find nearby servers holding its resource of interest. We note that it is often undesirable in large-scale networks to perform the placement using a centralized mechanism, which typically requires global topological information and suffers from a central point of failure.

In [12], we proposed a distributed algorithm to place replicated resources in peer-to-peer-type networks where each server is also a client. In this paper, we extend that work toward large-scale client-server networking systems. The key difference is that, while in [12], each node in the network must hold a replicated resource for itself and other nearby nodes, here we have two different types of nodes: servers holding replicas and clients demanding them. This different environment gives rise to the following main challenges that were not present in [12].

- The placement of each replicated resource must be performed in a manner that respects the needs of all clients' demands. For instance, two different game titles may be played by two flash crowds simultaneously in some region where there are only limited number of

[*]This material was supported in part by the National Science Foundation under Grant No. ANI-0117738 and CAREER Award No. ANI-0133829, and by a gift from Lucent Technologies. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

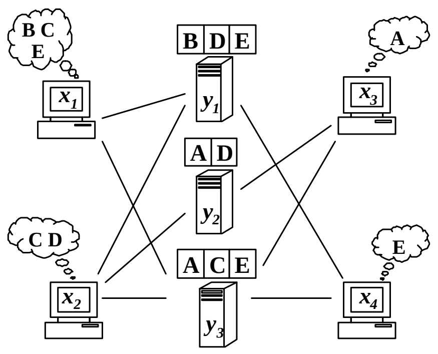


Figure 1: Example server-client network

server resources available. One must somehow determine which server should be allocated to each game to maximize "average" client performance.

- The placement mechanism must deal simultaneously with heterogeneous servers and clients; a server may host only a limited set of services. For example, some servers may have been designed specifically to host media streaming services and others specifically for Web services. Furthermore, some clients may be interested in only specific types of services.

In dealing with these challenges, we note that the performance of a replica placement mechanism, such as the distance between clients and replica servers, is in most cases determined locally (clients are interested in nearby replicas, not those far away). Our philosophy is to allow local decisions to be made in localized regions and avoid unnecessary interaction between nodes placed far apart from one another in the network. By doing this, we can improve the resilience and robustness of large-scale networking systems since the impact of unexpected failure and network dynamics is contained and can often be repaired quickly within the localized area.

The problem is explored in the context of server-client system, where *each server is assigned one of K colors and a color represents a specific replicated resource*, e.g., a specific game title, a specific web site. In our algorithm, each server decides its own color based on local distance information to the clients interested in its color. Preliminary simulation results show that our algorithm is scalable in terms of the number of servers and clients, and that the coloring achieved by the algorithm yields a distance between each client and its desired resources that quantifies significantly better than what resulted by randomly assigning colors.

The paper is organized as follows. In the following section, we present the system model and formally state the problem. We present our distributed algorithm in Section 3, and the performance evaluation results in Section 4. We briefly review related work in Section 5, and Section 6 concludes the paper.

2. MODEL AND PROBLEM

We consider a server-client network in which each server is capable of hosting some subset of replicated resources and each client is interested in some subset of resources. Figure 1 illustrates a simple example of such a system, where 5 different resource types, A through E, are to be replicated over a set of servers, $\{y_1, y_2, y_3\}$. In this system, each client has a different set of demands; For instance, client x_1 is interested in resources B, C, and E, while client x_4 is interested only in resource E. Similarly, a server may restrict the resources it hosts. For instance, server y_2 only supports replicated resources A and D, and can therefore only serve clients x_2, and x_3, but not x_1 and x_4.

We view the server-client system as an undirected, bipartite graph $G = (N, E)$, where N is the set of servers and clients and E is the set of non-negative weighted edges. N is further divided into two disjoint subsets, N_s and N_c, where N_s is the set of servers and N_c is the set of clients. We assume that each server and each client has a unique identity. An edge in E connects a client x and a server y. The edge's weight, $d(x, y)$, represents the distance between x and y. We assume that the graph G remains fixed during the execution of the algorithm.

Let $\Gamma = \{c_1, \cdots, c_K\}$ be a set of K colors, where each color in Γ represents a specific replicated resource type that a server can be assigned. A **server coloring**, or simply a **coloring**, $C : N_s \rightarrow \Gamma$, is a function that maps each server in N_s to one of K colors, where $C(y)$ is the color assigned to server y.[1] We further define $\Gamma_c(x) \subseteq \{c_1, \cdots, c_K\}$ as the set of colors that interest client x, and $\Gamma_s(y) \subseteq \{c_1, \cdots, c_K\}$ as the set of colors to which server y can be assigned.

A client x's **color distance** to color c_k, $d(x, c_k)$, is defined as the distance between x and the closest server of color c_k to x, i.e., $d(x, c_k) = \min_{y \in N_s}\{d(x, y) : C(y) = c_k\}$. $s_k(x)$ denotes the closest server of color c_k to client x. If there is no server of color c_k in the graph, then $d(x, c_k) = \infty$. Furthermore, let $d_2(x, c_k)$ denote the distance between x and the second closest server to x of color c_i. If client x is not interested in some color c_k, i.e., if $c_k \in \Gamma - \Gamma_c(x)$, then we say the color distance of x to c_k is 0.

Our goal here is to assign colors to servers such that the distance from each client to the closest server of each color can be made "as small as possible". This general problem can be posed more specifically in the context of one of several possible optimization functions. For example, one may want to minimize the sum of average color distances of all clients, or minimize the maximum color distances of all clients. However, none of these optimization problems necessarily satisfies our goal of placing multiple replica types such that the placement respects the needs of all clients. For instance, in online gaming networks, a game player's experience is influenced by the *maximum* delay between a game server and all clients that are connected to that server. In this case, minimizing the average (or sum) of the color distance does not improve the performance since, even if the average distance is small, some client may still be far from the server. Simply minimizing the maximum color distance does not solve the problem, either, since we are interested in reducing the distances to all replica types (not just the furthest one) from each client. Instead, we posit that a **lexicographically optimal** color allocation[3] best fits our problem, which is defined as follows.

Consider an l-dimensional ordered vector, $V = \langle v_1, \cdots, v_l \rangle$, where the v_i's, $i = 1, \cdots, l$ are ordered in decreasing order, i.e., $v_1 \geq v_2 \geq \cdots \geq v_l$. We say that V is lexicographically smaller than another ordered vector $W = \langle w_1, \cdots, w_l \rangle$ if

[1]Extending this model to the cases where a server is simultaneously assigned multiple colors can be easily done by mapping that server into multiple servers, each with a single color.

there is some $0 < i < l$ where $v_j = w_j$ for all $j < i$ and $v_i < w_i$. We also say that, for two arbitrary, unordered n-dimensional vectors V' and W', V' is lexicographically smaller than W' if the ordered version of V' is lexicographically smaller than that of W'. We write $V' \prec W'$ to indicate V' is lexicographically smaller than W', and $V' \preceq W'$ to indicate V' is lexicographically smaller than or equal to W'.

In the context of our problem, the lexicographically optimal coloring is defined as follows. Given G and a server coloring C in our model, consider an $|N_c|K$-dimensional vector $V(G) = (v_1, \cdots, v_{|N_c|K})$ whose components consist of all K color distances for all clients in the network, i.e., $d(x, c_k)$, $1 \leq x \leq |N_c|$, $1 \leq k \leq K$. Note that $d(x, c_k) \geq 0$ by definition. A lexicographically optimal color allocation, C_{OPT}, is one that assigns each server to a color such that the resulting vector $V_{OPT}(G)$ is lexicographically smaller or equal to any other vector $V(G)$, resulted by any other coloring C, i.e., $V_{OPT}(G) \preceq V(G)$.

The advantage of lexicographically optimal coloring is that it minimizes each color distance of each client, i.e., if the largest color distance cannot be reduced further, it seeks to reduce the next-largest color distance when possible. We can show that this problem is NP-hard by reducing an instance of 3-SAT to an instance of our problem. The proof of NP-hardness can be found in [13].

3. DISTRIBUTED SERVER COLORING

In this section, we describe our distributed algorithm that assigns colors to servers in large-scale, decentralized networks. Our approach is to let each server decide its own color based on the color distance information of nearby clients. In this environment, we must address several challenges that were not issues in [12].

- **Partial knowledge of topology :** The enormity of the network considered here implies that each client in the network can interact with only a subset of nearby servers to determine its distances to those servers. The distributed placement of replicated resources must be designed such that a global measure of color distances can be optimized by using only this local distance information.

- **Color distance inconsistency :** In our algorithm, each server decides and changes its own color based on the color distances of the clients, and clients' color distances change as servers change their colors. Keeping the consistency of color distance information between the servers and clients is difficult in the presence of network delay, which can result in servers' selecting their colors based on outdated client color distance information.

- **Servers oblivious of one another :** Since we assume that there is no special entity in the network that coordinates the communication among nodes in the network, each server's independent selection of its own color can impact another server's selection in numerous situations. For example, two nearby servers may simultaneously select the same color, c_i, without knowing each other's (same) selection, while it would have been better to have only one server of that color and have the other change to some other color in that region. A decentralized mechanism is needed to coordinate nodes in the network to avoid such undesirable configurations.

In what follows, we describe how we address these challenges with our distributed server coloring algorithm, beginning by describing the rule with which each server decides its color.

3.1 Coloring Rule

Let $S(x)$ denote the set of servers that interact with client x and could be assigned one of the colors of interest to x, i.e., $\Gamma_s(y) \bigcap \Gamma_c(x)$ is non-empty for any $y \in S(x)$. Similarly let $R(y)$ be the set of clients that interact with server y, i.e., $R(y) = \{x \in N_c | y \in S(x)\}$.[2]

Assume for now that each server y can obtain the color distances, $d(x, c_i)$ and $d_2(x, c_i)$, for all $x \in R(y)$ and for all c_i, $i = 1, \cdots, K$ through the color distance update mechanism described in Section 3.2.

Suppose a server y whose color is currently c_k changes its color to some other color c_j. We note that y's color change is "good" for some clients interested in c_j, but it can be "bad" for some clients interested in c_k, because we now have one less server of c_k and one more server of c_j after the change. More specifically, if y was the closest server of color c_k to x, i.e., $y = s_k(x)$ *before* y's color change, then $d(x, c_k)$ *increases* to $d_2(x, c_k)$. (If y was not $s_k(x)$, $d(x, c_k)$ does not change.) On the other hand, if y becomes the closest server of color c_j to some client x *after* y's change, $d(x, c_j)$ *decreases* to $d(x, y)$.

Now let $V_{k,l}(y)$ denote a $2r$-dimensional color distance vector whose components are the color distances to c_k and c_j of the clients in $R(y)$, i.e.,

$$V_{k,j}(y) = (d(x_1, c_k), d(x_1, c_j), \cdots, d(x_r, c_k), d(x_r, c_j)),$$

where $\{x_1, \cdots, x_r\} = R(y)$. We similarly define a "future" color distance vector, $V'_{k,l}(y)$, as

$$V'_{k,j}(y) = (d'(x_1, c_k), d'(x_1, c_j), \cdots, d'(x_r, c_k), d'(x_r, c_j)),$$

where $d'(x, c_k)$ and $d'(x, c_j)$ denote the distances to color c_k and c_j, respectively, that would result for client x if server $y \in S(x)$ would change its color from c_k to c_j.

With this color distance information, servers change their colors by the following color change rule.

Color change rule : Server y *decides* to change its color from c_k to c_l if

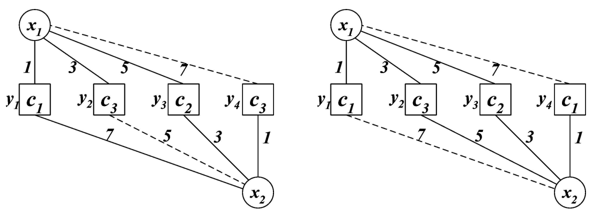
- y can be assigned c_l, i.e., $c_l \in \Gamma_s(y)$, and
- $V'_{k,l}(y) \prec V_{k,l}(y)$, i.e., $V'_{k,l}(y)$ is lexicographically smaller than $V_{k,l}(y)$.[3]

This change rule states that a server changes its color (1) if color distance of some client would decrease, and (2) if the increased color distance of any client after the change is smaller than the largest value among those color distances that would decrease. In other words, it ensures that

[2]There can be several factors that determine $S(x)$, such as the efficiency and scalability of server discovery mechanism, each client's capability to maintain the states and connections of servers, etc. However, this issue is out of scope of this paper, and we describe our algorithm such that it is orthogonal to this issue.

[3]If there are more than one such color, c_l, that satisfies the above properties, then y arbitrarily selects one of them.

a server's color change always produces a "better" coloring in a lexicographic sense.



(a) Before y_4's change (b) After y_4's change

Figure 2: Example of server color changes

In our distributed algorithm, servers continually change colors by the above color change rule. This color change process is illustrated in Figure 2, in which there are 4 servers, $y_1, \cdots, y_4$, and 2 clients, x_1 and x_2. The numbers along the lines connecting each client and each server represent the distance between servers and clients, and we want to assign servers one of 3 colors, c_1, c_2, and c_3. Consider a color assignment in Figure 2(a), where A solid line indicates $d(x_i, c_k)$ for each client and a dashed line $d_2(x_i, c_k)$. According to the color change rule, server y_4 can change from c_3 to c_1 since the change decreases $d(x_2, c_1)$ from 7 to 1 and increases $d(x_2, c_3)$ to 5 but not greater than the previous value, 7, of $d(x_2, c_1)$. Figure 2(b) shows the coloring after y_4's change, in which there is no server that can further decrease the color distance of any client without increasing some other distance beyond the largest distance that decreases.

In what follows, we proceed to the decentralized procedure used by servers and clients to keep the consistency of clients' color distance information and to coordinate the color changes of individual servers. Due to lack of space, we only provide a sketch of the mechanism; a detailed description can be found in [13].

3.2 Color Change Procedure

Servers and clients exchange messages to update new color distance information and to keep this information consistent between servers and clients. There are 6 types of messages: **Change**, **Update**, **Request**, **Accept**, **Reject**, and **Abort**. Here we briefly describe how and when these messages are used.

- **Request :** When, by the color change rule, server y decides to change its color from a color c_k to another color c_l, it sends this message to each client in $R(y)$, asking the clients to verify that the color distance information y used to decide the color change is the up-to-date color distances perceived by the clients. After sending this message, y waits for Accept or Reject messages from all clients in $R(y)$.

- **Accept** or **Reject:** A client x sends either of these two messages to server y as the response to a Request message from y. x sends an Accept message if the color distance information that y used in its decision to change its color is a valid one that reflects the up-to-date color distances perceived by x. Otherwise, x sends a Reject message to y.

- **Abort** or **Change :** Server y, which has issued a Request message, sends this message to the clients in $R(y)$ without changing color if it receives a Reject message from any client. On the other hand, if y receives Accept messages from all $x \in R(y)$, it changes its color and sends Change messages to the clients in $R(y)$, notifying that it has changed from some color, c_k, to another color, c_l.

- **Update :** Whenever client x perceives (by a Change message) its closest or second-closest server to any color has changed, it sends this message to each server in $S(x)$, notifying that some of its color distances has changed. This color distance information is used by servers when servers decide to change their colors.

Servers inform the clients of their new colors with the Change messages, and clients inform the servers of their new color distances using Update messages. Based on the color distance information carried in Update messages, each server decides whether or not to change its color using the color change rule. When a server does decide to change, it initiates a three-way handshake exchange of Request-Accept (or Reject)-Change (or Abort) messages to ensure that the color distance information it used to decide the color change is valid in comparison to the actual distances perceived by the clients. This three-way handshake mechanism overcomes any discrepancy of information in servers and clients that can occur because of the network propagation delay.

However, because servers can change their colors simultaneously without knowing other servers' changes, there can be cases where one server's intended color change conflicts with those of others, which can cause various race conditions. For example, suppose a client x has received requests from two nearby servers, both of which want to change to the same color. If x accepts both requests, then both servers will change colors and the process can livelock since they may keep changing their colors when they become aware of the other's new color. If both requests are rejected, then the process can again livelock, since both will re-attempt and possibly never be able to make a color change. If x simply holds the requests without sending Accept or Reject, then the process deadlocks – both will wait indefinitely.

We address this problem by introducing an order-based holding mechanism similar to what is used in [12] – clients hold requests from higher-ordered servers until the outcome of conflicting request of lower-ordered servers becomes available, whereas lower-ordered requests are immediately rejected if any conflicting request from higher-ordered server has been accepted.

We have proven that, by using color change rule in conjunction with the three-way handshake protocol and the holding mechanism, the server coloring converges to a state where no server changes its color. Its proof and more details on three-way handshake and holding mechanism can be found in [13].

4. PERFORMANCE EVALUATION

In this section, we present some preliminary results evaluated by event-driven simulations on top of Internet-like topologies generated by the BRITE Topology Generator[5]. We generate a transit-stub network consisting of 5,000 nodes (100 nodes/AS × 50 AS), and select servers and clients among those 5,000 nodes uniformly at random. We vary the number of servers, $|N_s| = 25, 50, 100$, and the number of clients, $|N_c| = 25, 50, 75, 100, 150, 200$. We also vary the number of colors $K = 5, 10, 15, 20$, and the number of servers contacted by a client $R = K, 2K$ for each value of K.

We use two different types of server configurations: **S-ALL** (each server is capable of all K colors) and **S-RAND** (each server is capable of a random subset of colors) Similarly, we simulate two types of clients' interests: **C-1** (each client is interested in one color selected at random) and **C-RAND** (each client is interested in a random subset of colors). Due to space limitations, here we present only some representative simulation results.

The distance $d(x, y)$ between a client x and a server y is set to the shortest-path distance in the transit-stub topology generated. We also use this distance to drive the message propagation delay between a server and a client, though our algorithm will converge for any message passing delay distribution between nodes. The average message delay between server and client is 50 msec.

At the start of the simulation, each client selects R closest servers among all servers. The simulation terminates when clients and servers cease sending messages, i.e., when servers cease to change colors. The measurement data was obtained by averaging out values from 10 simulations.

First, we evaluate the color distances in the coloring generated by our algorithm by comparing these distances to those in graphs where each server is assigned its color uniformly at random. We also present the color distances that could be achieved by the centralized version of the our algorithm, in which the placement is performed by an oracle that knows the color distances of all clients assuming that each client can reach all servers in the network. Note that the centralized algorithm is not optimal, and is essentially equivalent to the distributed algorithm when each client knows the distances to all servers. We show the results of the centralized algorithm to evaluate our distributed algorithm's performance when the interaction between servers and clients is limited within localized regions.

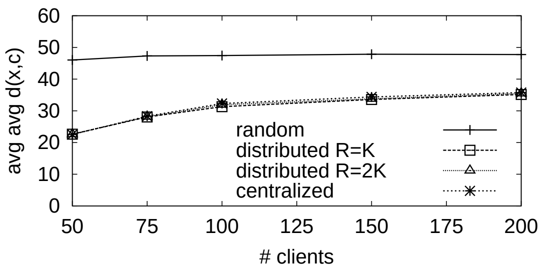


Figure 3: average color distances of clients ($|N_s| = 50$, $K = 20$, S-ALL/C-1)

Figure 3 depicts the average distance from each client to the colors of interest, i.e., $\text{avg}_{c_i \in \Gamma_c(x)} d(x, c_i)$, averaged over all clients x, when $|N_s|$ is 50 and K is 20. Each client is interested in a randomly-selected color, and each server can support all colors. We vary the number of clients along the x-axis. In general, we expect that the distributed algorithm would perform better than random coloring but worse than the centralized algorithm. We find that our distributed algorithm reduces the color distance of each client by 30 to 50 % compared to when servers are assigned their colors at random, and also that the distance is identical to the distance computed by the centralized algorithm for both values of R. This indicates that when each client is interested in only one color, our algorithm allows clients to find the color of interest in a nearby server by contacting only a small number of servers.

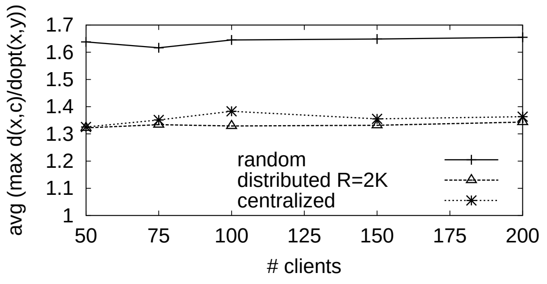


Figure 4: Normalized max color distances : $\frac{\max_{c_i} d(x,c_i)}{d_{opt}(x)}$ ($|N_s| = 50$, $K = 10$, S-RAND/C-RAND)

Figure 4 plots the distance from each client x to the *furthest* color among those in $\Gamma_c(x)$, i.e., $\max_{c_i \in \Gamma_c(x)} d(x, c_i)$, divided by the distance to the $|\Gamma_c(x)|$-th closest server, which we denote by $d_{opt}(x)$. $d_{opt}(x)$ represents the minimum distance that x must travel to reach all colors in $\Gamma_c(x)$, providing a lower bound of $\max_{c_i \in \Gamma_c(x)} d(x, c_i)$. We see from the figures that on average, our coloring algorithm achieves the distance to the furthest color from each client within factor of 1.3 from the minimum possible distance. This is quite an impressive result considering that, in many cases, there is no coloring in which each client x can reach all colors within $d_{opt}(x)$, in particular when nearby clients compete against each other over different sets of colors.

Now, we turn our attention to the transient performance of our algorithm by measuring the convergence time, that is, the time elapsed until the last message in the simulation is received by a node (server or client).

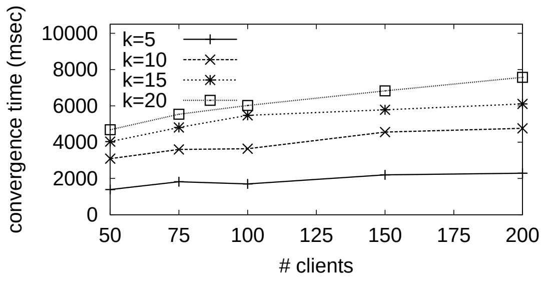


Figure 5: Convergence time ($|N_s| = 50$, $R = K$, S-ALL/C-RAND)

Figure 4 shows the convergence times (in msec) for different number of colors as the number of clients is varied along x-axis. We observe that the convergence time increases sub-linearly as we increase the number of clients. This suggests that the color change procedure is naturally parallelized across different regions of the network, especially when the size of the network is large. The increase of the convergence time as the number of colors increases also appears to be sub-linear (note that the gap between curves gets narrower as K becomes larger). This result suggests our algorithm is scalable not only in terms of the size of the network, but also of the number of colors.[4]

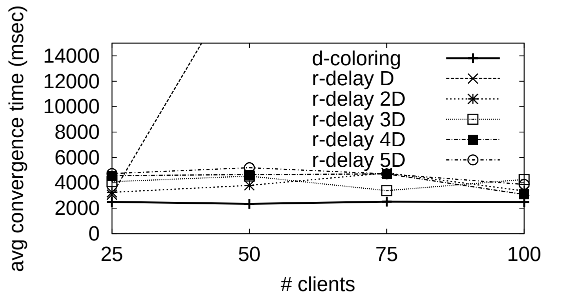


Figure 6: Convergence time : comparison to randomly delayed servers(S-ALL/C-1, $|N_s| = 25$, $R = K = 10$)

In Figure 6, we compare the convergence time of our algorithm to an alternative mechanism that solves race conditions by introducing a random delay in servers' color changes rather than utilizing the 3-way handshake. More specifically, if a server decides to change color, it does not initiate the 3-way handshake exchange of messages. Rather, it delays the color change and the transmission of Update message by a time interval picked exponentially at random. We use the average delay $\mu = D, \cdots, 5D$, where we arbitrarily set the constant D as the maximum distance between nodes in the network, in this case about 100 msec. We observe that our distributed algorithm always stabilizes faster than this alternative mechanism that uses randomized delays to break the synchronization. In particular, it is interesting to observe that it often takes a very long time for the random-delay mechanism to stabilize when the delay is relatively small (convergence time for $\mu = D$ is too high to appear within the y-axis range of this plot). This confirms our conjecture that independent operation of distributed servers will introduce race conditions, slowing down the stabilization process.

5. RELATED WORK

Our goal of assigning replicated resources to a set of server to minimize distances from the clients to nearby copy of resources is similar to that of optimal facility location problems[17] and of minimum k-center problems[8]. Though much work has been done in this area, including those in the context of web server replica placement[16], our goal differs significantly from these problems in that we allocate multiple resources (or "facilities") concurrently to a given set of locations, while the goal of facility location problems is to open up a single type of facilities in a subset of locations that minimizes the distance from the clients to the nearby facility. The work by Kangasharju et al[11] is probably the most closely related to ours. However, it solves the problem of placing replicas of arbitrary "objects" in nodes in CDNs using centralized algorithms, while we use on-line, decentralized algorithm that places replicated "servers".

Online gaming networks have recently drawn many researchers' interests, including gaming network architectures[7, 2], state synchronization techniques[7, 14], and scalable state management[4, 15]. Yet to our knowledge, there has been no distributed mechanism that addresses server placement problem in gaming networks. [10] presents an interesting observation on game players' preferences in selecting game server, which we can take into account when designing the server placement mechanism. We plan to do so in the near future.

[4]Though not shown here, we found the convergence time is somewhat insensitive to the number of servers.

6. CONCLUSION

We have presented a distributed, provably self-stabilizing algorithm that assigns server resources to replicated servers in large-scale server-client networks. We model the problem as a server coloring problem, where a color represents a specific replicated server type. In our algorithm, each server determines its own color based on local information obtained from nearby clients, and servers and clients co-operate in a distributed manner to help the server coloring stabilize. The simulation results show that the average distance between a client and the closest copies of the servers of interest to the client can be decreased by almost 50% compared to when the server replicas are randomly assigned.

7. REFERENCES

[1] Akamai. `http://www.akamai.com/`.

[2] D. Bauer, S. Rooney, and P. Scotton. Network infrastructure for massively distributed games. In *NetGames '02*, April 2002.

[3] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, 1992.

[4] A. R. Bharambe, S. Rao, and S. Seshan. Mercury: a scalable publish-subscribe system for internet games. In *NetGames '02*, April 2002.

[5] BRITE. `http://www.cs.bu.edu/brite/`.

[6] Counter-Strike.net. `http://www.counter-strike.net/`.

[7] C. Diot and L. Gautier. A distributed architecture for multiplayer interactive applications on the internet, IEEE networks magazine, vol. 13, no. 4, July/August 1999.

[8] T. F. Gonzalez. Clustering to minimize the maximum intercluster distance. In *Theoretical Comput. Sci. 38, 293-306*, 1985.

[9] Half-Life. `http://www.planethalflife.com/`.

[10] T. Henderson. Observations on game server discovery mechanisms. In *NetGames '02*, April 2002.

[11] J. Kangasharju, J. Roberts, and K. Ross. Object replication strategies in content distribution networks. In *Proceedings of WCW'01: Web Caching and Content Distribution Workshop, Berlin*, June 2001.

[12] B.-J. Ko and D. Rubenstein. Distributed, self-stabilizing placement of replicated resources in emerging networks. In *Proceedings of ICNP 2003*, November 2003.

[13] B.-J. Ko and D. Rubenstein. Distributed server replication in large scale networks. Technical report, Columbia University, March 2004.

[14] Y.-J. Lin and S. P. Katherine Guo. Sync-MS: Synchronized messaging service for real-time multi-player distributed games. In *Proceedings of ICNP 2002*, November 2002.

[15] H. Lu. Peer-to-peer support for massively multiplayer games. In *Proc. of INFOCOM '04*, March 2004.

[16] L. Qiu, V. Padmanabham, and G. Voelker. On the placement of web server replicas. In *Proc. 20th IEEE INFOCOM 2001*, August 2001.

[17] D. B. Shmoys, E. Tardos, and K. Aardal. Approximation algorithms for facility location problems (extended abstract). In *ACM Symposium on Theory of Computing (STOC)*, pages 265–274, El Paso, TX, May 1997.