Enforcing collaboration in a collaborative content distribution network

Namita Lal (154403)
Faculty of Sciences
Vrije Universiteit Amsterdam, The Netherlands

August 2007
Master's thesis, Computer Science

Supervisors:
Prof. Maarten van Steen
Dr. Guillaume Pierre

vrije Universiteit amsterdam

ENFORCING COLLABORATION IN A COLLABORATIVE CONTENT DISTRIBUTION NETWORK.......... 1

1 INTRODUCTION.......... 3

2 RELATED WORK.......... 8

2.1 CDNs AND COLLABORATIVE CDNs.......... 8
2.1.1 BURSTY WORKLOADS.......... 10
2.2 MECHANISM DESIGN.......... 12
2.3 ITERATED PRISONER'S DILEMMA AND TIT-FOR-TAT.......... 13
2.4 BITTORRENT.......... 14

3 SYSTEM DESIGN: ENFORCING COLLABORATION IN THE CCDN CONTEXT.......... 16

3.1 FIXED NEIGHBORHOODS.......... 17
3.2 ACCOUNTING.......... 18
3.2.1 PAIR-WISE TRADING SCHEME AND TIT-FOR-TAT WITH FORGIVENESS.......... 18
3.2.2 JOINING THE SYSTEM.......... 21
3.2.3 BUILDING TRUST.......... 21
3.3 VALIDATING CLAIMS.......... 24
3.3.1 MOBILE IPV6 AND TCP HANDOFFS.......... 24
3.3.2 VALIDATING THROUGH RANDOM CHECKS.......... 25

4 SYSTEM IMPLEMENTATION.......... 28

4.1.1 NEW APACHE SERVER MODULE STRUCTURE AND MODERATOR PROCESS.......... 29
4.1.2 STRUCTURE OF PERIODIC REPORTS EXCHANGED BETWEEN NODES.......... 29
4.1.3 ACCOUNTING INFORMATION AT EACH SERVER.......... 30
4.1.4 BANDWIDTH THROTTLING.......... 31

5 EVALUATION.......... 33

5.1 TEST SETUP.......... 33
5.2 CLIENT-SIDE WORK LOAD GENERATION.......... 34
5.3 RESULTS.......... 38
5.3.1 SINGLE SERVER PERFORMANCE WITH AND WITHOUT COLLABORATION.......... 38
5.3.2 SERVER PERFORMANCE WITH FREE RIDING.......... 40
5.3.3 SERVER PERFORMANCE WITH FREE RIDING UNDER ENFORCED COLLABORATION.......... 43

6 CONCLUSIONS.......... 47

REFERENCES.......... 48

1 Introduction

Cooperative peer-to-peer (P2P) systems are overlay networks of nodes that rely on the resources contributed by the participating nodes themselves; such systems allow geographically distributed nodes to come together for some mutual benefit such as content sharing, resource sharing, etc. P2P architectures differ from the traditional client-server systems in that the latter rely on the resources of a few server nodes that are used to serve the clients of the system. Pure P2P networks however, do not have the notion of clients and servers, instead such networks consist of "peer" nodes that are both the clients and the servers of services in the system. Such networks are useful for many purposes, such as information sharing, multicasting, sharing bandwidth resources, etc. However, an important criterion for the proper functioning of cooperative P2P networks is that the nodes in the network contribute their resources (bandwidth, storage space, and computing power) properly so that as more users join the system, the collective strength of the system to serve these users increases as well.

In practice, however, it is seen that in the absence of incentives to contribute resources, users act selfishly (rationally) and exploit the system without making any contribution themselves (a notion that we will refer to as free riding henceforth). This kind of user behavior is extremely detrimental to the performance of the system due to the decrease in the resources available to the users as a whole. It also produces strain on the few good nodes in the system which further decreases the incentive to contribute positively. Thus the benefits of a collaboration decreases tremendously as more and more nodes resort to free riding (a phenomenon often referred to as the tragedy of the commons in economics [2]). Additionally, free riding also affects the robustness of P2P networks, throwing the system back in to a client-server mode, where few nodes act as the servers of the resources and the performance of the system as a whole depends on the performance of these few nodes.

In the past, many file-sharing P2P networks that implemented their protocols with naïve assumptions of voluntary cooperation have shown to have a large percentage of free riders. For example, studies on Gnutella have revealed that about 70% of the nodes in the system contribute no files at all, and nearly 50% of all responses are returned by the top 1% of sharing hosts [1]. Hence today most of the peer-to-peer systems implement some mechanism to enforce the nodes to co-operate with each other and force them to contribute positively to

the collaboration. Such mechanisms have been developed by file-sharing systems such as Kazaa, BitTorrent, eMule, etc., that keep track of node behavior, and reward or punish them accordingly. These efforts seem to have paid off in the sense that they have reduced the number of free riders and have motivated nodes to become cooperating members of the network if they wish to avail the services provided by other users in the collaboration.

The mechanisms implemented by these P2P file-sharing systems for enforcing collaboration essentially work by holding the nodes accountable for their resource usage and their resource contribution, and can be roughly classified as reputation-based or trade-based mechanisms. Reputation-based schemes use a rating system to assign points to peers in the network based on the past behavior of the node. This rating is then used by other nodes to determine the suitability of the peer during interactions. Usually, users with low reputation rating are allowed access to fewer resources, are mistrusted and find their transactions restricted. On the other hand, trade-based schemes try to limit resources consumed by the users using micropayments or pair-wise trading mechanisms. Although many such solutions exist in this space, most of them are very specific to file-sharing networks, where the following assumptions hold:

(i) Collaboration can be locally fair, i.e. fairness is local in time. Services offered by one node to another needs to be immediately compensated for in order to carry the transaction forward in time. For instance, in BitTorrent – a P2P file sharing protocol – peers exchange fragments of files that they are interested in. A server uploads *k* fragments to its peer and then waits for the peer to reciprocate with fragments that it is interested in. If no such packets are forthcoming it 'chokes' the connection to that peer by temporarily refusing to upload to it. Thus, the service provided by a peer needs to be paid back immediately in time for the transaction to move forward and for it to get more file fragments from the peer node.

(ii) The second assumption is that nodes in the system are the direct consumer of services provided by their peers and hence they can easily verify the parameters of the services provided by each other. Again, in the case of BitTorrent, peers exchange fragments of files they are interested in, thus they can verify the fragment contents themselves and the rate at which the data was uploaded to them by a peer. This directly establishes the service quality provided by a peer node.

However, there are many scenarios of peered systems in which these assumptions do not hold and thus the existing solutions will not be able to serve the purpose of enforcing collaboration. Systems where the services offered by the nodes are not temporally related (as in assumption (i) above) require special considerations to be taken into account. For example, bandwidth sharing systems such as in 2fast [9] require nodes to remember the contribution of collaborating peers. Peers may not need reciprocal service immediately but only at a later point in time. Similarly, there may be peered systems in which the service provided by the nodes is consumed by clients that are not part of the collaboration. For example, in a collaborative content distribution network (CCDN) consisting of peered Web servers, the services of the servers are consumed by Web clients that are not participants of the collaboration.

The peered systems described above require two special considerations as far as cooperation is concerned: (1) a collaboration mechanism that is globally fair, i.e., over time the system is fair to all nodes providing them resources in proportion to the resources they contributed; and (2) with the additional requirement that the consumers of the service may not be the peers directly, and hence there is a requirement for proper monitoring schemes to be in place in order to verify node behavior correctly.

This thesis addresses the problem of enforcing collaboration in the context of CCDNs. CCDNs present a scenario in which both the assumptions of local fairness and direct behavior ratification of nodes do not exist. These networks are an effective technique for Web content providers to increase their quality of service by pooling their resources and serving their collective set of clients together -- without increasing the cost of hosting, hardware, or bandwidth. The contents of an origin server (a server that provides a site) are replicated across the collaborating servers; HTTP requests for the origin site may be redirected to the collaborating servers depending on several parameters such as the network-based proximity of the client to the servers and the load on the servers. However, like any collaborative system, collaborative CCDNs face the problem of providing correct incentives to the participating servers to contribute their resources correctly. For instance, in the absence of a mechanism to enforce good behavior, a Web server may redirect all its clients to the peer servers, utilizing their bandwidth resource without reciprocating this assistance and refusing

to serve the peer's clients in return. Thus, there is a requirement to enforce fairness in such a system.

Moreover, if a surrogate server provides content to its peer's client, it may not have an immediate requirement for the peer's resources (it may not have clients requesting data at that moment in time), thus the assistance provided by a peer must be remembered over time and returned when the need arises. Also, since the clients (the immediate consumer of the services) are not directly aware or part of the collaboration, the origin server cannot easily ascertain that a peer indeed serviced its clients for the claimed data volume or at the claimed rate. This brings in the necessity to verify a peer's contribution by additional means.

Along with the assumptions stated above, we also introduce an additional requirement for the framework: there is no global picture of behavioral characteristics available to the nodes in the system. In other words, they must act based on information available to them locally. Such an assumption is important to create protocols that have low overhead and are scalable across large distributed systems. The goal is therefore to provide a protocol:

(i) that is fair on the nodes in the system, i.e., the benefits that nodes receive from the system is in proportion to their contributions to the system;
(ii) the implementation of which has low overheads and does not impinge on the costs of the functional protocols;
(iii) that is resilient enough to take into account many free-riding scenarios (such as nodes behaving nicely in the beginning, only to turn into free riders later on).

This work proposes a solution to the problem of enforcing collaboration between nodes in a peer-to-peer content distribution networks by employing a mechanism similar to the BitTorrent tit-for-tat protocol. Based on the observations made by Axelrod [3] we know that cooperation in a collaboration system can emerge only when nodes can (1) recognize other nodes (in order to punish or reward node behavior); and (2) when there is certainty that nodes have repeated connectivity. Based on these, we propose to fix the overlay of nodes participating in a CCDN in such a way that mutual inspection of behavioral history becomes possible. Such a neighborhood structure ensures that the servers recognize each other, and have repeated transaction with each other, the conditions essential to ensure accountability. Further, [3] also states that the tit-for-tat strategy is most effective in the iterated prisoner's dilemma (IPD) game and that cooperation between rational nodes may evolve as a

consequence of implementing this protocol. By relating the transactions between the nodes in a CCDN as an IPD game, we implement a tit-for-tat protocol between them. Each peer node in a neighborhood maintains an account for every other neighbor, recording the services provided by the peer, its bad behavior, etc. This accounting information is used to enforce fairness both for the data volume as well as the data rate exchanged between the peers. The protocol ensures that free riders cannot exploit the nodes in the system by establishing an upper threshold for the loan that a node is willing to provide to its neighbors (loan is the difference in the bytes uploaded for a peer's clients by a node and the bytes the peer has uploaded for that node's clients). As we will see later this mechanism is a low-overhead, resilient implementation of the tit-for-tat strategy, which in turn provides a global equilibrium condition in which cooperation can evolve in the system.

Further, when the origin server's clients are redirected to a peer we also need to verify the service provided by the peer node to these clients. In order to solve this problem we employ random checks that can be conducted by the origin server using TCP socket migration over IPv6. If we assume that the connection between the servers and the clients are over IPv6, then a server that needs to verify a peer's claims can ask the peers to hand over the client back to itself (using socket migration and IPv6 handovers). Once it has reconstructed the connection, it can create a temporary raw socket on the connection and use the acknowledgement number received from the client to verify the number of bytes that have been transferred over the connection. Since the origin server has information about the time at which the client was handed over to the peer (all redirections go through the origin server), the origin server can also verify the data rate contribution. Both these aspects allow the origin server to verify a peer's claims fairly accurately. By performing this check at random points in time, there will be a constant pressure on the peers to provide accurate reports.

The rest of the thesis explains the details of the problem introduced in this section and the solution proposed here. Section 2 describes the related work in this field that is important to understand the motivation and effectiveness of our solution. Section 3 describes the design of our solution in detail. Further, Section 4 gives the description of an example implementation of the proposed protocol and finally, Section 5 describes the performance evaluation experiments that we performed and the results that we collected from them.

2 Related work

Before we delve deeper into the solutions presented in this work, it is important to understand a few concepts that form the basis on which the solutions are built upon. In Section 2.1 we first study the characteristics of CDNs and CCDNs, their benefits and the problems of creating an incentive mechanism that system designers face in deploying CCDNs in the real world. We then turn our attention to the field of mechanism design (MD) which can be used for developing strategies for building incentives systems for collaborative systems in Section 2.2. In Section 2.3 we delineate one such mechanism design -- the tit-for-tat strategy that has proven to be very effective as an incentive mechanism for the BitTorrent file sharing protocol which in turn is described in Section 2.4.

2.1 CDNs and collaborative CDNs

Content Delivery Networks (CDNs) are a very effective technique for content providers to move away from the centralized mechanism of Web content delivery. In the centralized scheme, each Web client connects to a single or a cluster of Web servers that field(s) the requests of all the clients. CDNs replace the traditional centralized infrastructure and provide several (edge cache) servers sitting at the edge of the Internet between the user and the origin Web server, redirecting clients to suitable edge caches as need be. Essentially, CDNs are composed of a content delivery infrastructure, a request routing mechanism and an accounting mechanism, that together provide a better service to their users by routing their requests to an "edge cache" close to the end users.

Figure 1 shows the conceptual architecture of a typical CDN, where many replicated web servers are located at the edge of the network closer to the Web clients, and content requests are routed to the closest replicated Web server instead of the origin server. Thus, they provide many benefits to the origin Web server like decrease in client response time (by redirecting requests to the edge server closest to the client), better bandwidth and resource provisioning at the origin server's side, protection from flash crowds, etc.

Many commercial as well as academic solutions have been implemented in this field, Akamai and Mirror Image are chief among the commercial solutions. On the other hand, Globule [10] and Coral CDN [21] are examples of *collaborative CDNs*, which are

collaborations consisting of geographically distributed Web servers, servicing their collective set of Web clients together. Similar to the edge-cache scenario, the requests for one server can be redirected to another depending on several parameters such as the load on the systems, the proximity of the servers to the client, etc. Such collaborations provide a mechanism for the participating servers to trade temporarily unused resources between each other which can lead to better utilization of the pooled resources.

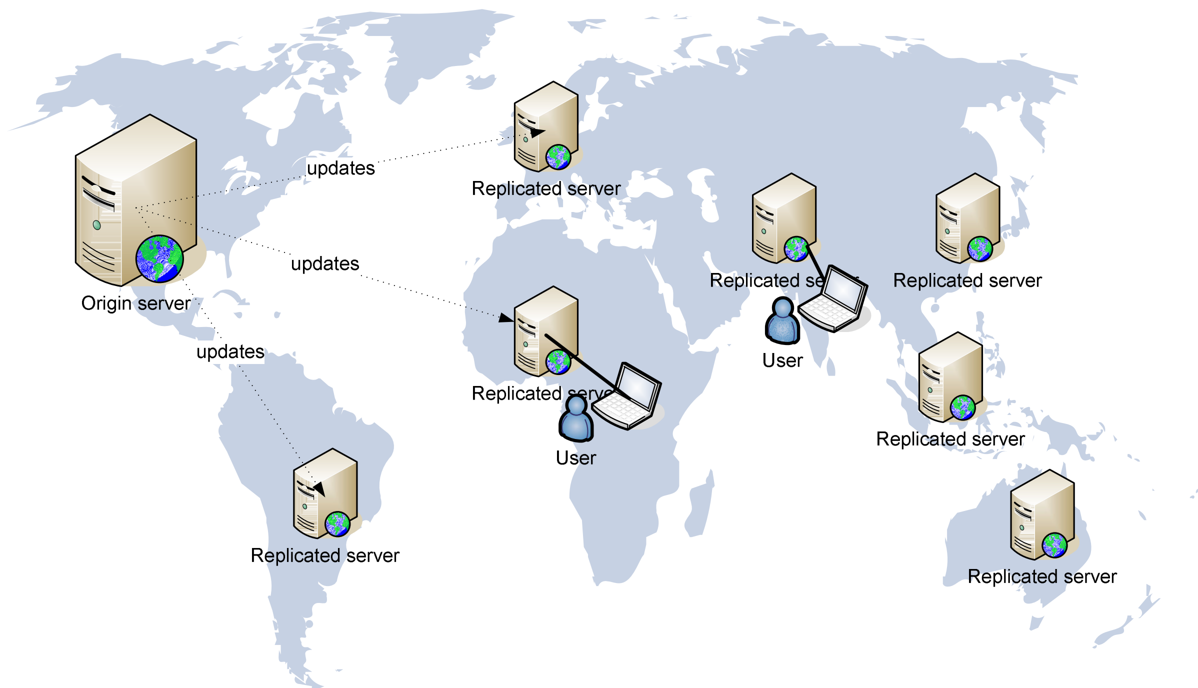


Figure 1 Conceptual architecture of a Content Delivery Network.

Despite these benefits, there are many obstacles on the path to effective deployment of CCDNs: (i) the problem of trusting the integrity of the data served by the peers to foreign clients, (ii) the problem of misbehaving nodes creating denial-of-service attacks for the clients of the nodes in the collaboration, (iii) the problem of being exploited by the peers to serve data on their behalf without the peer nodes reciprocating similar services, (iv) the problem of verifying a peer's Quality of Service (QoS) to clients, and other such malicious and free-riding misbehaviors. In this work we look at the last two problems and try to provide solutions to prevent the problem of free riders in the system. We do not look into the aspects of deliberate malicious behavior launched by peer nodes to jeopardize the data or service provided by good nodes in the system.

2.1.1 Bursty workloads

One of the important factors that make collaborative CDNs an effective model to serve clients is that the rate of arrival of HTTP requests at the servers is bursty. Figure 2 illustrates Web request burstiness over several time scales. This phenomenon can be explained as follows: Requests to the servers are generated by a huge number of clients each with its own think time characteristics – the time between two consecutive requests from the same client. Furthermore, the distribution characteristics of this think time has a large amount of variance, and additionally the think times of different clients are not independent of each other due to human user's wake and sleep patterns. Also, due to the scheduled publication of Web content, requests arriving at a server seem to be correlated. Several studies have revealed a self-similar property of network traffic in large distributed systems such as the WWW [4]. This self-similar property of the underlying traffic manifests itself as a bursty characteristic of the request rate over several time scales. The upshot of this is that the request traffic arriving at a Web server can come in bursts whose peak rate is 8 to 10 times the average rate seen by the server. For example, studies have shown that HTTP request traffic contains bursts observable over four orders of magnitude [6].

Such bursts in the traffic rate can temporarily overload a server beyond its capacity and lead to a large degradation in the QoS seen by the clients. As shown by [5], even a small amount of burstiness in the request arrival rate can degrade the performance of a Web server considerably, leading to large client response times. Studies have shown that the reason why the performance of Web servers goes down when the client load experiences bursty behavior is because under such conditions the queues of critical server resources are more susceptible to rapid buildups, which result in higher rates of request rejections [14][17].

We know that the load demand on the individual Web servers is highly variable in the presence of bursty request arrival rate, and that burstiness has a strong negative impact on the performance of Web servers. In such a scenario, provisioning high bandwidth at the server can be wasteful, and the servers are better off offloading requests to each other for a short time when they observe a burst in the traffic rate. Hence, by participating in a CCDN the servers can be shielded from the performance degradation caused by such bursts without provisioning extra bandwidth. Moreover, if the servers were to provision more bandwidth individually, it would be a waste of resources when the bursts are absent in the traffic.

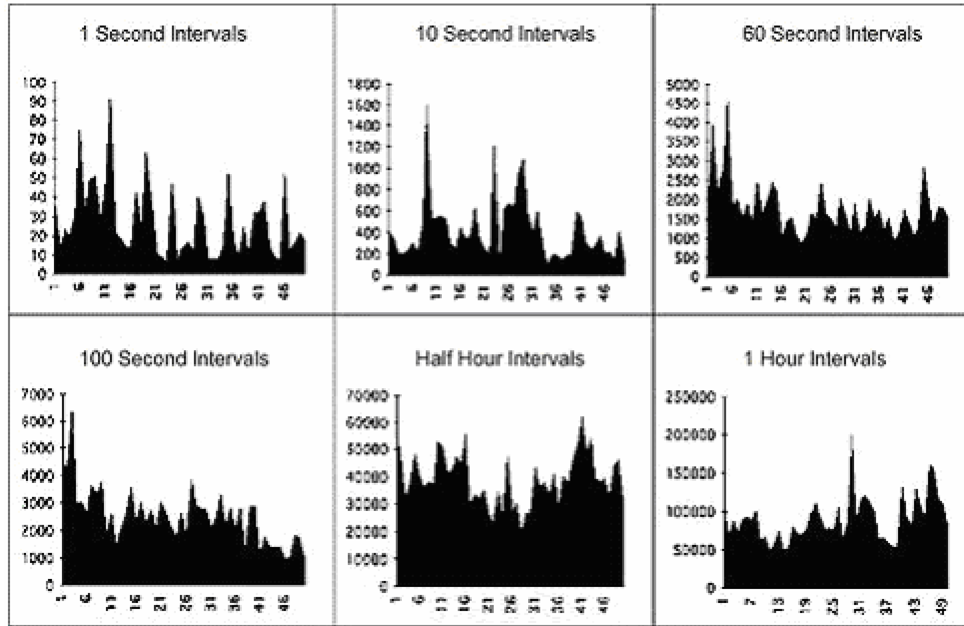


Figure 2 The figures show that the number of requests arriving at a Web site reveal high variability or burstiness, peak rates during bursts may exceed the average rates by factors of five to ten, momentarily putting the system in overload conditions [7].

Thus, CCDNs can help servers deal with bursts of high load by offloading requests to peer servers that are not facing bursts of their own. Eventually, the collective set of clients of the servers would see a better quality of service even though no individual node had to provision more upload bandwidth capacity.

However, the big problem is that most of the CCDNs that exist today work on the naïve assumption of voluntarily cooperating servers. As discussed before, such assumptions do not work in real-world scenarios where participating nodes have selfish traits (at the very least), and thus, lead to ineffective deployments. For example, although the ultimate goal of the CoralCDN is to be able to run many third party nodes within it, due to security related considerations, as of now CoralCDN only runs on centrally administered 'trusted' nodes [20]. In order for these CCDN systems to be successfully deployed and utilized in a more decentralized manner, we need a mechanism to impose cooperative behavior between participating Web servers. In order to construct an efficient solution to this problem, we first gather the basic requirements that must be fulfilled:

(i) We need to build a system that forces the nodes to co-operate with as low overhead as possible.

(ii) Since we are dealing with large-scale distributed systems, nodes in the network will not have a consistent global picture of the whole system and must act on the basis of some local information available to them.

(iii) The system should be defined in such a way that the strategies open to a node to maximize its benefit should also be beneficial to the system.

(iv) The incentive system should be fair on the nodes, i.e., the benefits that nodes receive from the system are in proportion to contributions they make to the system.

It so happens that these characteristics are very similar to the requirements for solutions in the field of mechanism design (MD), and its distributed, algorithmic version -- distributed algorithmic mechanism design (DAMD). We look at the concepts of this field below, and specific solutions developed along the lines of DAMD in the sections that follow.

2.2 Mechanism design

As discussed in the introduction, many solutions have been proposed to tackle the selfish behavior of users in P2P systems. We are interested in a specific scheme called mechanism design (MD), which are schemes in which it is assumed that nodes are rational and play strategies to maximize their benefit. Certain "rules of the game" are defined so that the system shows positive performance behavior even with rational users pursuing selfish strategies. Formally, a mechanism is a specification of possible player strategies and a mapping from the set of played strategies to desired outcomes. Mechanism design can be thought of as inverse game theory – where game theory reasons about how agents will play a game, MD reasons about how to design games that produce desired outcomes [15]. As we can see, the problems that MD looks to solve are very similar to our problem of ensuring node collaboration in a CCDN. We too need to ensure that the strategies open to a node in the system to maximize its benefit are also the ones that maximize the benefit of the system as a whole. However, many aspects of classical MD are computationally infeasible. The field of algorithmic mechanism design (AMD) and distributed algorithmic mechanism design (DAMD) address the computational issues of classical MD. The latter assumes that the mechanism is carried out via a distributed computation. The aim of this work is to create a

well-designed mechanism that will provide incentives to the selfish nodes in a CCDN to behave in a correct way so as to increase the overall benefits of the system.

2.3 Iterated prisoner's dilemma and Tit-for-tat

One mechanism design to ensure cooperation is the tit-for-tat strategy between nodes. As mentioned earlier, Axelrod [3] discusses the properties of tit-for-tat protocol being the best deterministic strategy in the iterated prisoner's dilemma (IPD) game [22]. This strategy has been employed in the BitTorrent protocol to great benefit.

In game theory, prisoner's dilemma is a type of non-zero-sum game in which two players may each "cooperate" with or "defect" (i.e. betray) the other player without prior knowledge of the strategy of the other player, the only concern of each individual player ("prisoner") is maximizing his/her own payoff, without any concern for the other player's payoff. Suppose that two players are involved in a PD game, whose payoff assignments are as follows: T (temptation to defect), R (reward for mutual cooperation), P (Punishment for mutual defection) and S (Sucker's pay off). If player 1 co-operates and player 2 defects, player 2 gets T points and player 1 gets the Sucker's playoff S. If both cooperate they get R points each, and if they both defect they get the Punishment for mutual defection P. Furthermore, in such a game the following inequality holds for the value of the payoff assignments: $T > R > P > S$. Note that given this assignment, a player always benefits by defecting regardless of what strategy the other player adopts. Thus, all rational players will choose to defect, even though they could earn a mutually higher reward if both decided to cooperate.

In iterated prisoner's dilemma (IPD) however, the game is played repeatedly. Thus each player has an opportunity to "punish" the other player for previous non-cooperative play. Cooperation may thus arise in equilibrium, since the players now have to worry about future outcomes that depend on the opponent's reaction to their current strategy. The incentive to defect is overcome by the threat of punishment, leading to the possibility of a cooperative outcome. However, for cooperation to emerge out of an IPD game the following inequality must hold in addition to the one stated previously:

$$2R > T + S$$

In a further experiment, Axelrod ran IPD tournaments between players with different strategies, with the assumption that players could recognize each other and maintained

behavioral history about each other, and he discovered that tit-for-tat was the simplest and the best strategy in IPD games. This strategy can be summarized as follows:

(i) Cooperate in the first interaction
(ii) Thereafter defect or cooperate according to what the other player chooses

Additionally, he also states the following conditions must hold in order for the strategy to be successful:

(i) Be nice, never defect first, always cooperate until and unless your opponent defects,
(ii) Retaliate bad behavior, the node with some high probability must return bad behavior, this prevents nodes from being exploited and discourages free riding,
(iii) The node must be forgiving, as in it must retaliate so long as the opponent defects, but cooperation must again be reciprocated with cooperation.
(iv) Clear behavior, i.e., the semantics of a node's behavior must be clear to its opponent.

Thus, by relating the interactions between the nodes in a CCDN as an IPD game and then implementing the tit-for-tat strategy, we know that there is a possibility for introducing cooperation in the system.

2.4 BitTorrent

BitTorrent is a P2P file sharing communication protocol forming an excellent example where a tit-for-tat like strategy has already been employed to enforce cooperative behavior between nodes. The protocol works as follows. Data files in BitTorrent are divided into equally sized fragments. Peers exchange these fragments to build the complete file. Peers that wish to download a file using this protocol contact a "tracker" that tells them from which peers they could download the pieces of the file they are interested in. A group of peers connected to each other in order to exchange pieces of a file is called a swarm. If the swarm consists of initial seeders only (a node that has the complete file) the node can request file pieces directly from them. As more peers join the swarm the peers begin trading pieces with each other based on a tit-for-tat protocol.

This is done by choking (not uploading to) and unchoking (uploading to) a peer node. Each peer maintains the current download rates from all the peers it is connected to. Using this information, it unchokes (starts uploading to) a set of 'u' connections with the highest

download rate. All other connections are choked; however, a mechanism called opportunistic unchoking allows peers to look for a good link periodically. If during the process of opportunistic unchoking a node finds a peer with a rate higher than one for the current unchoked connection, the new node replaces the old slower connection.

This concept of choking and unchoking depending on the service received by the peers builds an incentive mechanism in the swarm for nodes to upload the file fragments at good rates to the peers if they want to download file fragments at a good rate as well.

Although the BitTorrent protocol works well in the file-sharing scenario, such a protocol cannot be directly applied in an environment such as CCDNs since in such systems the service provided by a peer does not need to be immediately remitted in time for the transaction to move forward. In a CCDN environment nodes would need to remember a peer’s contribution so that it can be compensated for at a later point in time when the peer actually needs the node’s services. Also, unlike the file-sharing scenario, the nodes in a CCDN are not the direct consumers of the services provided by the peers, which makes accounting for a peer’s contribution a challenging problem.

3 System design: Enforcing collaboration in the CCDN context

In CCDN systems with rational nodes, system designers encounter a node's conflict between the benefit of offloading its own client requests to other peers in the network and the cost incurred in having to serve its peer's Web clients. In the absence of any accounting mechanisms, most nodes will prefer to offload their requests to their peers without giving service to the clients of their peers in return, i.e., the most beneficial strategy of these rational nodes is to free ride the system. In the context of MD, we need to design a strategy or an incentive mechanism that forces the peers to cooperate by making that strategy the one that brings maximum benefits from the system. This can be achieved by a simple mechanism of rewarding nodes that contribute resources to the system and punishing those that free ride it. As a study of evolution of collaboration in a system, Axelrod [3] observed that cooperation can only emerge when nodes can (1) recognize other nodes (in order to punish or reward node behavior) and (2) when there is certainty that nodes have repeated interactions. Based on these observations, we propose to fix the overlay of nodes participating in the CCDN in such a way that mutual inspection of behavioral history becomes possible.

As described in Section 2.3 we can build a low-overhead protocol to provide fairness to nodes in a CCDN system by relating the scenarios in an IPD to the interactions that may occur between peers in such a system. We can then designing a tit-for-tat strategy for their interactions. The advantages of such a mechanism are manifold:

(i) It fits well with our requirement that the nodes must interact with each other and choose strategy based on local information, and thus has low overhead.

(ii) We believe that tit-for-tat can be applied for a solution that requires globally fair behavior between nodes.

(iii) The architecture of a fixed overlay scenario where we assume that the nodes can recognize each other and have repeated interactions satisfies the premise laid down by the solution research.

Building such a system requires three main mechanisms. First, in order to ensure the requirement that the nodes in the system recognize each other and have repeated interactions with each other, we need to make sure that the peers that a node interacts with are fixed. There is thus, a necessity of fixing the neighborhoods in the system. Additionally, in order to

prevent Sybil attacks (which we explain below) it should not be easy for nodes to jump between such neighborhoods. We address this by building a central overseer of the structure of such groups and their memberships. We discuss these mechanisms in Section 3.1.

Second, within a group, peers need to implement the tit-for-tat strategy to make sure they are not exploited by their neighbors. This brings us to the requirement that the peers should maintain some accounting information about each other and take actions based on that information. A node will need to account for a peer node's contribution in terms of data uploaded to the redirected clients, and the rate at which this data was uploaded. It can use this information to decide its own strategy based on how good or bad the peer has been. Details of this accounting information are explained in Section 3.2.

Finally, since the direct consumers of the services of the system are outside the information loop, our system must also verify a node's contribution without blindly relying on the reports sent by the peer. Section 3.3 discusses the details of such a mechanism, which uses the concept of random check using socket and IP-level migration of network connections between the servers to verify each other's behavior.

3.1 Fixed neighborhoods

In this section we will look at the principles of organization of nodes in our system. One of the main problems in P2P systems is that of Sybil attacks [8]. In one form of such attacks, nodes can subvert the reputation or accounting mechanisms in the system by leaving and rejoining the system using different identities after exploiting its resources. A system's vulnerability to such Sybil attack depends on how easy is it for attackers to generate new identities.

In order to prevent this form of Sybil attacks, we first assume that the nodes have a fixed overlay. When a new node joins the system it is assigned to a neighborhood by a central overseeing server. The function of this overseer is restricted to assigning and keeping track of neighborhood structures in the system (in that sense the system we are proposing is a hybrid peer-to-peer system, with at least one node that has some 'super' peer powers).

Without such an overseer entity, users may exploit nodes in a neighborhood group for their benefit and move onto other groups without returning the services of the current group. Since

our model relies on maintaining and acting on local information, such an exploitation of the system would be possible.

Furthermore, creating fixed neighborhoods is also important from the perspective of maintaining repeated interactions between the nodes, so that they can recognize each other and are able to make mutual inspection based on the history of interactions with each other. This is an extremely important criterion without which it would not be possible to implement a proper tit-for-tat strategy between the nodes.

3.2 Accounting

This section details of the accounting mechanism that implements the tit-for-tat scheme between the nodes in a CCDN. We first look at the basic accounts maintained by each node about its neighbor. Then we present the mechanism implemented by nodes to handle long-term interactions with their peers. Long-term good behavior is rewarded in the system using a notion of trust which allows larger and larger units of data to be traded between trusting nodes. This trust mechanism also allows nodes to detect and punish a previously trusted peer that could turn into a free rider after a long series of successful trading interactions. Such a mechanism provides the incentive for the nodes to maintain good behavior over long periods of time since the rewards are made proportional to the parameters of successful trading interactions between them.

3.2.1 Pair-wise trading scheme and Tit-for-tat with forgiveness

As explained, CCDNs work by redirecting client requests to peer Web servers depending on several load and latency parameters. Peers handle such redirected requests on behalf of the origin server and expect similar assistance from the origin server when they need it. In order to build fairness into the system we must ensure that no node can offload client requests to its peers indefinitely without serving requests on behalf of the other nodes in the system as well. Additionally, we must also ensure that not only is the number of data bytes served on behalf of the nodes equivalent but also that the rate at which the data is served to the clients is the same. Since the overlay neighborhood is fixed and there is only a fixed set of nodes to which a node can offload requests, we can introduce a very simple, yet effective scheme to ensure fairness by employing a pair-wise tit-for-tat strategy at every node. Normal interactions

between nodes in the network will proceed as follows. Each node maintains the following two sets of accounts for each node in its neighborhood.

(i) A set of accounts for the volume of data exchanged with its peers

This set of accounts will primarily consist of two variables, the cumulative bytes served by a peer on behalf of the node and the cumulative bytes served by the node on behalf of a peer node. We call these variables `CumulativeBytesConsumed` and `CumulativeBytesServed`, respectively. The difference between these two variables for each peer will give information about their deficit. As long as the deficit of a peer is positive or is negative but within a certain bound, its clients' requests will be honored, i.e. client requests will be handled on behalf of a peer as long as the following is true:

$$\text{CumulativeBytesConsumed} - \text{CumulativeBytesServed} < \text{Deficit Threshold}$$

This relationship can also be stated as follows: at any point, the maximum number of data bytes that a node must serve its peer is:

$$\text{Peer's credit} + \text{Deficit Threshold}$$

Where $\text{Peer's credit} = \max\{0, \text{CumulativeBytesServed} - \text{CumulativeBytesConsumed}\}$

The deficit threshold itself can either be a constant or a function of a "Trust" factor between the nodes. How this trust factor can be built between the nodes is discussed later in Section 3.2.3. When a node sees that a peer's credit has exceeded the maximum allowable limit, it stops servicing that peer's client until the credit is built up again (i.e. the peer has contributed its own resources in serving the nodes clients).

(ii) A set of accounts for the data rate exchanged with the node

In addition to the accounting information regarding the volume of data contributed for and by a peer, in order for the mechanism to be globally fair, we also need to keep track of the rate at which this data was served to the respective clients. In the absence of such an accounting mechanism, peers do not have the incentive to serve foreign clients (redirected clients) at good data rates. Imagine a scenario consisting of two nodes A and B. At some point during their interaction node B sees a high burst of client requests and redirects its clients to node A. It is eventually in a debt of 1MBytes. At a later point node B has to pay back this debt to node A. However, node B is aware of its load pattern and knows it does not require node A's assistance in the foreseeable future. In such a scenario, node B does not have the incentive to

serve node A's clients at a high rate since it can build its credit back with node B even with a slow rate and in time for its next requirement for redirection. This leads to an ineffective strategy for node A, since its clients could be served at a very low data rate, decreasing their QoS. Essentially, since the service provided by nodes cannot be traded instantaneously, we need to remember more parameters to qualify the quality of service provided by the peers.

We thus arrive at the notion of maintaining the data rate that was provided to and by the peers in addition to the data volume contributed by them. It may be possible to keep a fine-grained record of how much data was served at what rate - and reciprocate such behavior. However, such a mechanism would be highly cumbersome. As fluctuations in client access rates and Internet related performance variability can lead to extremely variable data rates served even by well-behaving nodes. Therefore, a fine-grained accounting mechanism will produce an extremely strict system with no tolerance at all. In order to build more tolerance into the system each peer maintains two rate trends instead: a trend for the rate at which data was served to its client by the peer (`RateConsumedTrend`), and a trend for the rate at which it served data to the peer's clients (`RateServedTrend`). These trends can be maintained by calculating the exponential weighted moving average (EWMA) by periodically assessing the current data rate:

$$\text{RateConsumedTrend}(t) = \alpha\text{RateConsumed}(t) + (1 - \alpha)\,\text{RateConsumedTrend}(t-1)$$

$$\text{RateServedTrend}(t) = \alpha\text{RateServed}(t) + (1 - \alpha)\,\text{RateServedTrend}(t-1),$$

Where $0 < \alpha < 1$, and t is time at which the trends are calculated.

The weight parameter α can be tweaked to change the importance of past behavior vs. the current behavior of the nodes. If α is high (say 0.9) then the trends are dominated by a node's current behavior, and the past history of the node is almost ignored. Similarly, if α is low (say 0.2) then the opposite is true: the current behavior of the node is ignored in favor of its past history. In order to conform to the observations for proper functioning of the tit-for-tat protocol laid out in the previous section, we need to make sure that the nodes are retaliatory as well as forgiving, thus the value of the weight parameter should be kept high enough so that the node's current behavior is reciprocated, while at the same time some information from its past behavior is reflected in the current decision as well to forgive temporary bad service.

Note that these trends do not decay with time: the rates are calculated only as long as at least one of the peers is serving foreign clients for the other. These trends will be used to affect the data rate assigned to the peer in the following manner:

If $\frac{\text{RateConsumedTrend}}{\text{RateServedTrend}} < \text{rate_limit}$; Then punish the peer by decreasing the rate assigned to serve clients of the peer by a fixed amount (in this case the `deficit_limit` of the peer will also be decreased as described in the next section):

$$\text{RateAssignedToPeer} = \text{RateAssignedToPeer} - \delta,$$

If $\frac{\text{RateConsumedTrend}}{\text{RateServedTrend}} > \text{rate_limit}$; Then reward the peer by increasing the rate assigned to server the clients of the peer by a fixed amount:

$$\text{RateAssignedToPeer} = \text{RateAssignedToPeer} + \delta,$$

where ‘`rate_limit`’ is a configurable parameter in the system between 0 and 1. If ‘`rate_limit`’ is set to 1 then the system is extremely exacting since it reacts to even the smallest difference in data rates. If the parameter is set too low, it is make the system becomes open to exploitation by making it too tolerant.

The parameters of the system must be chosen such that there is tolerance in the system so that well-behaved nodes are not punished due to factors that degrade the service that are beyond their control.

3.2.2 Joining the system

When a new node joins the system, no existing peer will serve its clients since the new node does not have any positive account with them. In order to start transactions a new node must provide an initial credit to the existing nodes that are part of its neighborhood. Apart from kick-starting the new nodes, this also thwarts Sybil attacks by introducing a cost of joining the system.

3.2.3 Building trust

The trust level between the nodes in the system increases as they interact positively with each other. This increase in trust level can be translated into larger deficit limits between peers in

order to improve the efficiency and benefits of the system. Larger deficit thresholds allow servers to handle large bursts more effectively by offloading requests to the peer. At a later point when the burst has subsided, the node can reciprocate the contribution back to its peer. Furthermore, it is possible that due to external problems even well-behaving nodes may not be able to service requests temporarily in a proper manner. For example, the node may experience temporary congestion, network outage, etc. Building a notion of trust between nodes that have been interacting for a long time allows more leeway in such situations.

In order to build trust between peers, we need a mechanism to account for how good a node has been during its history of interactions. Trust can be built by measuring many different types of parameters such as the ratio of the number of successful interactions to the total number of interactions with a peer node, or the cumulative number of bytes contributed by the peer, or by relying on the number of times a node refused service even though it was in debt. The trust factor of a node controls its deficit limit. High trust factors allow the node to take larger loans from its peer and thus deal with larger bursts without running out of credit. On the other hand, if the trust factor of the node is low, its deficit limit is lower (or even zero) and its peers are reluctant to take any risks by loaning bandwidth to it.

By feeding this information back into the accounting system, we can increase the deficit limit of well-behaving peers so that the peers with which a node has had a long-standing interaction benefit from having maintained good behavior over a longer period of time. This in turn provides incentive for nodes to be cooperative in an extended manner. The deficit limit of a badly behaved node will be zero, i.e. no debts will be granted to the node unless there is some positive contribution provided by it to begin with. As the node starts interacting with a peer, its deficit limit increases or decreases as a function of its behavior in the following manner:

(i) For every "x" bytes contributed the deficit limit is increased by $\frac{x}{increase_param}$,

$$deficit_limit = deficit_limit + \frac{x}{increase_param},$$

where $increase_param > 1$. This can also be interpreted as the peer getting a reward for every 'x' number of bytes it contributes. Thus, the deficit limit will increase as

long as the peer contributes data to the node. (In our experiments we used the value 100 for `increase_param`).

(ii) In order to prevent a peer from behaving well initially to build a high deficit limit and then defecting to bad behavior, we will decrease the value of the deficit limit considerably, each time the node behaves badly. A few examples of bad behavior on the part of a node include:

   a. Refusing redirecting requests even though the peer is in debt. These are direct refusals in which the peer server does not accept a client redirection; this can be detected by the origin server if the peer does not respond to an IPv6 handover.

   b. A situation when the trend in the data rate provided by the peer is below a certain lower limit. These are indirect refusals: once the peer accepts a redirected client it can provide an extremely low data rate to the client, such a case it considered bad behavior as well.

In such scenarios we decrease the deficit limit in the following manner:

$$\text{deficit_limit} = \text{decrease_parameter} * \text{deficit_limit},$$

where $\text{decrease_parameter} < 1$. (In our experiments we used the value 0.1 for `decrease_param`).

(iii) We record the number of times a peer's `deficit_limit` has reached zero, and if it happens too often (this is a configurable parameter to the system), the node can stop communicating with (boycott) the peer altogether, marking it as a node with which an extended relationship cannot be built. In such a case the node can ask the overseer to assign it a new neighbor.

This accounting information is stored in permanent storage, so a node remembers a misbehaving peer over time. This is useful when the misbehaving node tries to leave and rejoin the system in a bid to wipe its history. Since we have kept the accounting information local between each pair of interacting nodes, the effect misbehavior related information is restricted. It is only applicable if a free-riding peer tries to leave and rejoin a neighborhood that consists of a node that boycotted it previously. In order to make sure that a misbehaving node cannot change its neighborhood too many times to escape boycott, we need to make sure that the overseeing server is informed

about local boycotts in a neighborhood. The overseer already has some basic information about all the nodes in the system (the neighborhood structure). In addition to this, it could maintain a flag for each node indicating whether the node has been boycotted by any of its neighbors. This information could be used to disallow the node from switching neighborhoods or leaving and rejoining the system subsequently.

(iv) If a peer fails a random check, it is considered a breach of contract. The node boycotts it and seeks out a new neighbor from the overseeing server.

3.3 Validating claims

The additional difficulty in deploying a CCDN is the problem of validating a peer's contribution claims. The direct consumer of the service provided by the collaboration and hence the entity that is capable of making accurate statements about a node's contribution is the Web client that actually receives content from the participating servers. However, since CCDNs try to make their functioning as transparent as possible, the clients are not aware of the existence of the collaboration between the participating Web servers. Thus, they are out of the information loop in the system, and data about node behavior cannot be extracted from them. In such a scenario, it is very easy for a node to claim that it has served 'x' number of bytes to foreign clients at high data rates without actually having done so. Thus, without a mechanism to verify node behavior, the accounting system, trust values between nodes etc., are all ineffective for preventing free riding in the system. In the following sections we outline a solution to this problem based on IPv6.

3.3.1 Mobile IPv6 and TCP handoffs

In Mobile IPv6 each node has a permanent "home" address (HA), which acts as a permanent identifier of the node, as well as, a temporary "care-of" address (CA), which identifies the node's current location. Mobile IPv6 ensures that network traffic sent to the home addresses is transparently forwarded to their care-of counterparts. To this end, it relies on clients communicating with mobile nodes to translate between home- and care-of addresses. In this way, the node can move around and have different "care-of" addresses while at the same time receive traffic addressed to its home address.

In order to redirect clients between Web servers and also to allow transparent random checks about a node contribution, we adopt a mechanism very similar to the one used in [16] where the authors employ Mobile IPv6 techniques to create a versatile Anycast system that decouples the logical addresses of mobile nodes from their physical location. In our model of CCDNs, we assume that the communication between client and server nodes is on top of mobile IPv6, and assign a logical "home" address to each server to which its clients send Web requests. We can now redirect the clients of the origin Web server transparently to any of its peers by handing them over to the peer and treating the peer's nodes as the "care-of" addresses of the origin server. This can be done using the IP handover mechanism provided by mobile IPv6.

Since the HTTP protocol usually runs on top TCP, when handing over client HTTP connections between peer servers, it is not sufficient to hand over just the IP part of the connection. This is because with TCP and HTTP on top, some transport and application layer state is being maintained both at the client and the server side, and in order for the handovers to be completely seamless, we need a mechanism for handing over TCP socket states from one peer server to another without affecting the state of the connection at the client. Thus, apart from utilizing mobile IPv6 mechanisms to switch between different IP addresses, we need to transfer the TCP state of the connection between the servers as well. As discussed in [16] we can implement TCP-level handover using tools such as 'tcpcp'[19]. The handoff mechanism itself requires the exchange of about 90 bytes of TCP connection state between the peer servers, and the whole process of switching mobile IPv6 address and TCP state can be optimized to about 236 ms in the worst case on a 1.5Mbps link (~200ms to transfer the TCP state + ~36 ms for the IPv6 handover). This brings the cost of performing a random check and handing the socket connection back to the peer to about 236 + 36 ms (since the peer already has the socket state, we do not need to transfer it back to the peer). Thus, the overhead of this method is low enough to not to degrade the client-perceived performance, and can be employed to ensure proper node behavior in the collaboration.

3.3.2 Validating through random checks

In order to solve the problem of validating the claims of a peer, we implement a random checking mechanism based on IPv6. This mechanism utilizes the separation of the logical

address and the actual physical address of a node in Mobile IPv6. As discussed before, this decoupling allows transparent switching of IP addresses of a mobile node so that the node is accessible at the same logical address, even though its physical address has changed. Our solution uses a combination of IP handovers and TCP socket state migration and implements a mechanism in which Web client connections can be transparently switched between the peer servers without affecting the client side state.

The mechanism works as follows. The Web clients always initiate requests with the origin server. Upon receiving requests the origin server will make a decision to redirect the client to a peer server (based on its load, or proximity of the client to the peer server or the peer server's account) and may hand over the client to the peer. For every client that was redirected to a peer, the origin server will maintain information that includes the timestamp at which the client was handed over to the peer. As the client starts interacting with the peer server, the origin server will receive periodic reports from the peer about the data volume and the data rate contributed by the peer. In order to prevent the peer node from lying about its contribution, the origin server can make random checks which will proceed as follows:

(i) Ask the peer server to hand over the client connection back to the origin server. Such a handover will be completely transparent to the clients.

(ii) Since TCP is a reliable protocol it keeps track of the sequence numbers (sender's side) and acknowledgement numbers (receiver's side) of the data bytes exchanged between the sender and the receiver. The origin server will quickly create a raw socket, and query the client.

(iii) The browser will in turn send out the correct acknowledgement number for this connection, which also indicates the number of bytes it has received on this connection.

(iv) The origin server can now verify the peer's claim for data volume and rate (since it knows the time when the client was handed over to the peer) that it has contributed.

(v) If all is good, the client will be handed back to the peer server.

As we will discuss later, this random check mechanism has a low overhead and thus, will be effective in making sure that the peers do not lie to each other about their contributions. Figure 3 summarizes the conceptual flow for the random check mechanism.

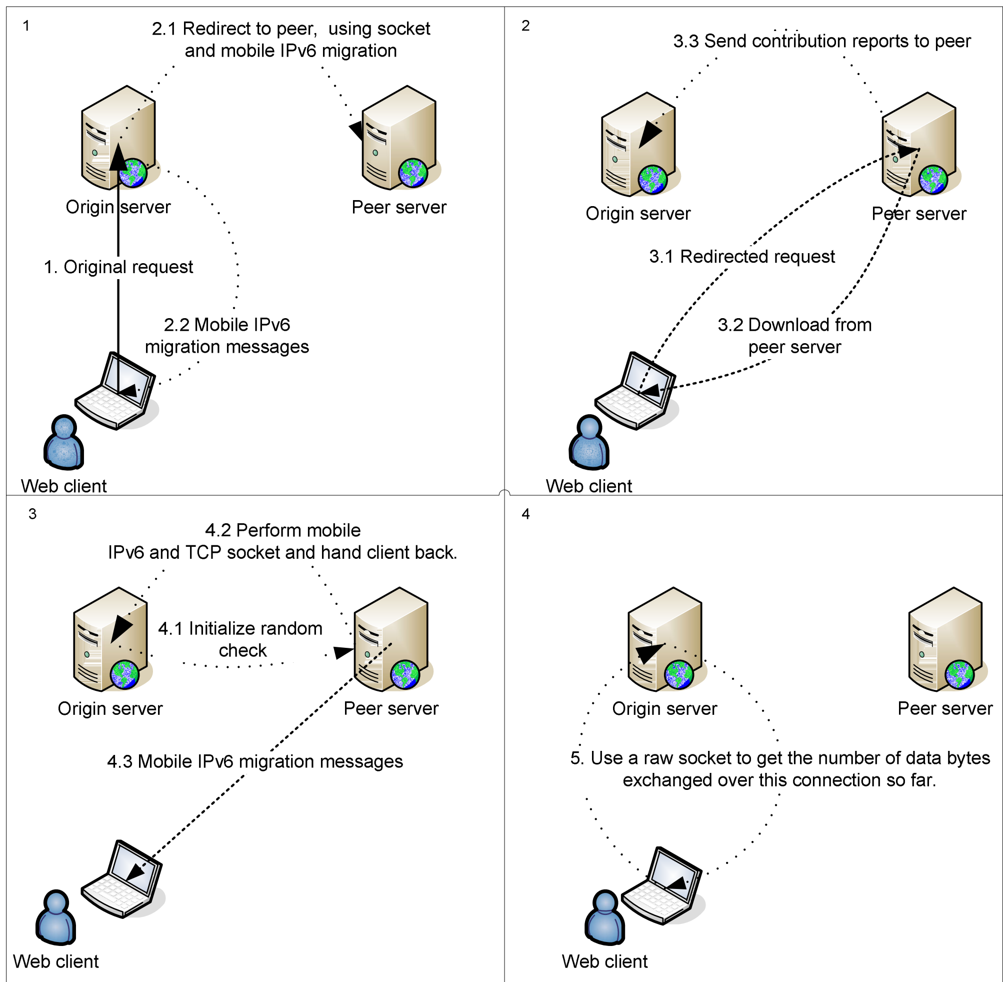


Figure 3 Summarizes the conceptual flow used for the random checking mechanism.

4 System Implementation

In this section, we describe an example implementation of the design described so far. Figure 4 illustrates an overview of our implementation. We extended the Apache 2.2 Web server by writing a new module that takes care of the collaboration aspects of the server. An origin server can redirect its clients to its peer server. In such a case, the peer server will check the account of the redirecting server before it honors the requests of the redirected clients. If the accounting information allows it, the peer server starts handling the requests of the redirected client and sends periodic reports to the original server about its contribution in terms of data volume and date rate. The original server records this information on permanent storage (hence peers remember each other's behavior over time). This example implementation is specifically built with the emphasis on measuring the effectiveness of the proposed solution, and is not complete in terms of real deployment of the framework. Aspects such as joining the system, obtaining new neighbors after boycotting a peer server, membership management, neighborhood construction and IPv6 random checks where left out in the interest of evaluating the performance of the core algorithm.

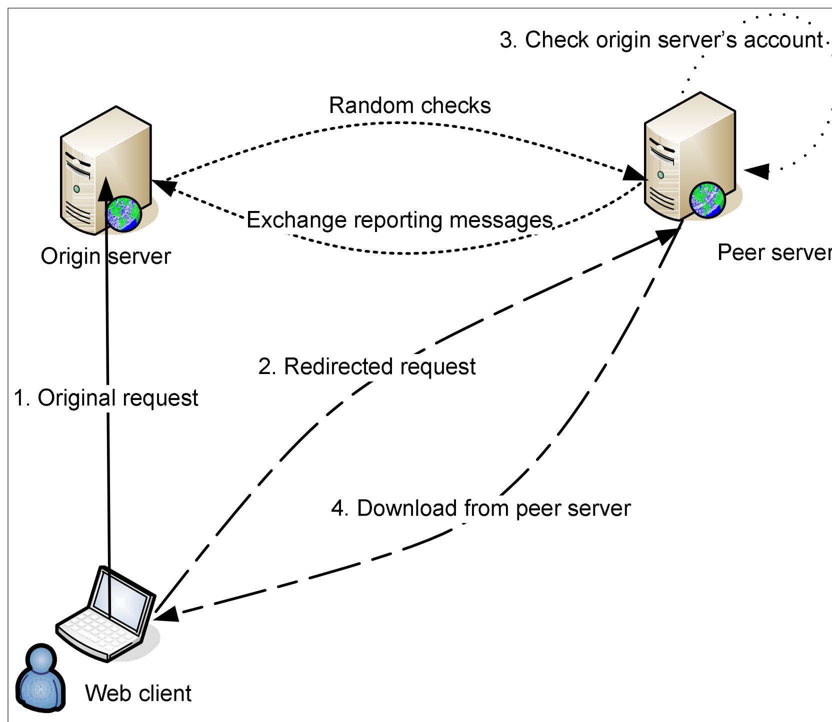


Figure 4 An overview of the architecture implemented for our experiments

4.1.1 New apache server module structure and moderator process

Figure 5 illustrates the details of the implementation of the collaboration mechanism for each Web server. As the very first step, the pages of each Web server must be replicated at the peer server so they can serve clients redirected from the origin server. A collaborator process is started next which establishes a TCP connection with each neighbor's collaborator process. The collaborator processes will exchange contribution reports on this connection. The collaborator process also creates a new network traffic shaping class in the node's kernel for controlling the outgoing rate of traffic for the clients of each peer. Both servers redirect traffic to each other in a round-robin manner using HTTP redirection; every alternate request is redirected to the other server. Next, we implemented a new module for the Apache Web server that inserts its hook in the name translation phase of processing HTTP requests. When a request is received, the module checks whether the target request URL is for an object that belongs to a peer server. If this is the case, it marks the request as a foreign client, checks whether the accounting system allows the request to be honored, if so it adds the client's IP address to the network traffic class that is assigned to the peer server and starts uploading the requested file to the foreign client. The module also sends periodic reports to the collaborator process about the number of bytes uploaded to the foreign client and the rate at which the data was uploaded. The collaborator process in turn, updates the accounting information about the peer, recalculates its deficit, its deficit threshold and trust parameters, and changes the rate assigned to the network class for the peer based on this information. The collaborator now consolidates the reports and sends them over to the peer. The collaborator process also receives reports about contribution from the peer and performs similar accounting updates.

4.1.2 Structure of periodic reports exchanged between nodes

The periodic reports exchanged between the peers have to be designed in such a way that they survive temporary node unavailability. When contribution reports are sent to the origin server it is possible that the receiving node is unable to receive them temporarily and the reports may be lost. Lost reports mean that valid contribution may be lost forever from the accounting system. For such cases, we need to make a provision that a lost report does not mean lost contribution. Thus, we designed the reporting structure such that each report gives the cumulative number of bytes contributed towards the peer's clients. Thus, even if reports

are lost, subsequent reports will add the correct contribution to the peers account at the origin server.

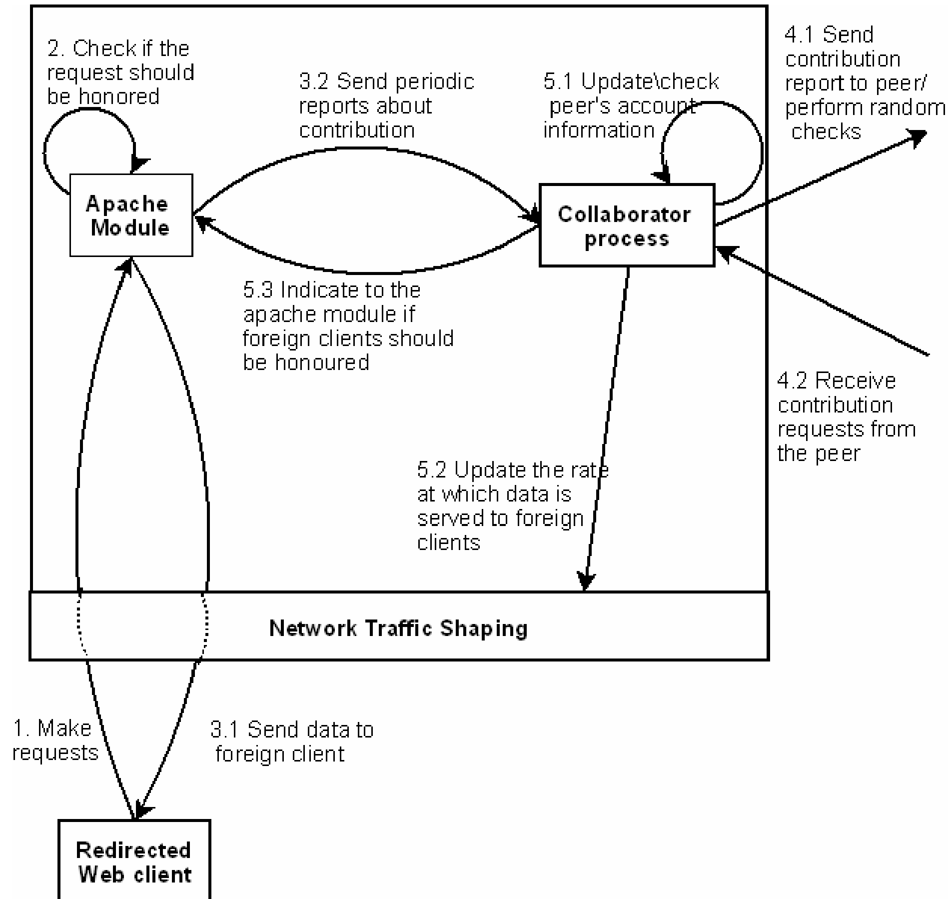


Figure 5 Details of the implementation of the collaboration mechanism in a Web server

4.1.3 Accounting information at each server

Table 1 shows the accounting information recorded by a node for each of its neighbors. The record consists of two parts: a long-term section that is stored in permanent storage and recorded over a large period of time, and a fine-grained section that is transient and may be kept in memory and discarded over time. A permanent record consists of the overall contribution of the peer, the cumulative data volume and rate trends, the (mis)trust factor built up over the course of the interactions. The rationale behind keeping this information relatively permanent is to prevent Sybil attacks, and keep track of misbehaving nodes that leave and rejoin a system after a while. The transient information in the accounting records includes the status of a node's current clients that have been redirected to a peer and the peer's foreign client that a node is handling. Other data that are stored by a node includes

configurable parameters of the system such as the rate_limit parameter that is the lower bound on the data rate provided by the peer, and the aging parameter that is used to weigh the calculation for volume and rate contributions.

Name of the peer, IP, identification information		
For each client handed over to this peer	- Timestamp of last report - Timestamp of handover - Bytes contributed for this client	Transient
For each client being handled for this peer	- Timestamp of last report - Timestamp of handover - Contribution towards this client	Transient
Random checks	Next random check timestamp	Transient
Peer accounting information	- CumulativeBytesServed , CumulativeBytesConsumed - RateConsumedTrend, RateServedTrend, RateAssignedToPeer - DeficitLimit - Boycott flag	Permanent
Configurable Parameters	- Aging parameter α (for trend calculation) - Rate_limit parameter (for capping the rate limit) - Deficit limit Increase_parameter (for trust calculations) - Deficit limit Decrease_parameter (for trust calculations) - Rate assigned increase\decrease parameter (for trust calculations)	Permanent

Table 1 Accounting information maintained at every node

4.1.4 Bandwidth throttling

Apart from maintaining an account and constraining the number of data bytes contributed towards peers (using the tit-for-tat mechanism), the nodes must also restrict the rate at which data is served to peer's client. In order to apply such a mechanism for data rate, the collaborator process uses a Hierarchical Token Bucket (HTB) packet scheduler to shape the data traffic served to the peer's clients [18]. During initialization, the collaborator process creates:

(i) A higher-level CCDN class that will shape all traffic that is part of the collaboration, i.e., a class that shapes all outgoing traffic for foreign clients that belong to any peer in the system. This class will be bounded to a maximum bandwidth limit 'L' that a node wishes to contribute to the collaborative CDN as a whole.

(ii) Within the higher-level CCDN class, the collaborator module will create a subclass for every peer in its neighborhood. Initially these classes will be unbounded, meaning they can share 'the bandwidth limit 'L' equally. However, as relationships are built and peers start redirecting traffic to each other, this bandwidth bound on each class

will reflect the rate assigned limit 'l' to the peer depending on its behavior trend and accounting mechanism as discussed in section 3.2 This will bound the maximum bandwidth contributed towards the peer's client to this limit. Figure 6 illustrates this mechanism.

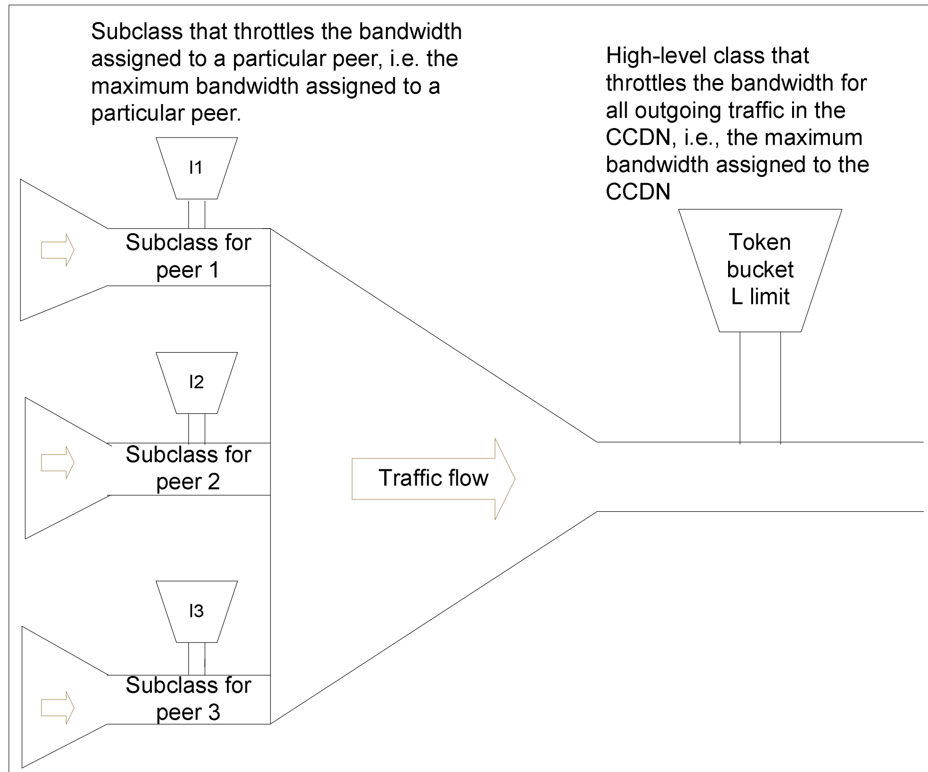


Figure 6 The HTB mechanism implemented to ensure outgoing data rate accounting

We need to investigate further the case where the same client is browsing multiple sites concurrently. Since the filters to classify traffic to different HTB classes are based on the destination IP address only, this mechanism will break if the client is browsing multiple sites whose servers have different accounts with their neighbor. Here we will need to investigate how we can differentiate between the requests of the same client for different sites and filter the traffic accordingly so that different requests are served at different QoS in accordance with the accounts of the corresponding peer node.

5 Evaluation

This section describes the experiment we conducted to observe the performance and the behavior of our incentive scheme. We begin by describing the test setup, the client side and server side software, the WAN emulation that we set up between the clients and the servers, and then we present the experiments we conducted and the results that were obtained.

5.1 Test setup

For the testing environment we used the Distributed ASCI Supercomputer 2 (DAS-2) [6]. DAS-2 is a wide-area distributed cluster of 200 Dual Pentium-III nodes designed by the Advanced School for Computing and Imaging (ASCI). The DAS-2 consists of five clusters, located at five Dutch universities. The cluster we used contains 72 nodes and runs RedHat Linux. Each node contains:

• two 1-GHz Pentium III CPUs
• at least 1 GB of RAM
• a 20 GByte local IDE disk
• a Myrinet interface card
• a Fast Ethernet interface (on-board)

Figure 7 shows the basic architecture of our test bed in which we use a NISTnet emulator to simulate a WAN-like environment between our client and server machines. We have two Web servers in the test bed, each running Apache 2.2 and our collaboration architecture as described in the section above. Further, we emulated a wide-area network (WAN) among the servers and the clients by directing all the traffic between them to an intermediate router which uses the NISTnet network emulator. This router delayed the packets sent between the different machines to simulate a wide-area network. The router was configured to simulate asymmetric ADSL connections with server-side download bandwidth of 2Mbps and upload bandwidth of 240 Kbps. Also, the link latency was configured to 100ms between all machines.

Four client machines were used to generate the request workload for the servers. These machines generate Web requests using the S-Client mechanism [5]. We looked at several

synthetic Web traffic generators such as httperf [13], apacheBench, etc, but we found the design of S-Client most suitable for generating highly bursty request rate in order to study the effect of collaboration between servers in the presence of bursty traffic. We divided the client machines into groups of two, and assigned them to generate work load for each of the servers separately. The exact method for the load generation is described in the following section.

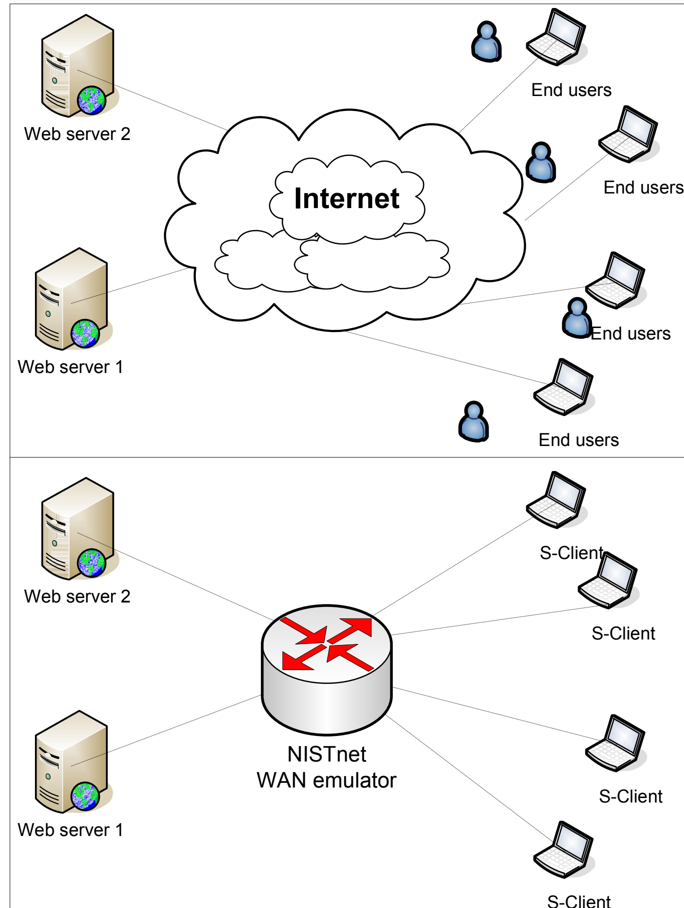


Figure 7 The schema illustrating the NISTnet setup used in our experiments

5.2 *Client-side work load generation*

As mentioned above, the client load generation mechanism is based on the S-Client architecture. Many workload generators assume a closed system model, where new job arrivals are triggered only by job completions. However, as explained in [5] if closed system models of request are used then as the load on the server is increased the request generation rate at the client side goes down as well. The clients are thus unable to produce request rates larger than the capacity of the server and hence, fail to create large bursts in the request rate.

For our tests we need to generate bursty client-side traffic, with peak rates about 8 to 10 times higher than the average rate in order to temporarily put the server into overload conditions. Although real Web servers can easily be overloaded by the almost infinite number of potential clients on the Internet, it is difficult to simulate such conditions in laboratory environments with closed system models of request generation. What is required is a method of generating requests where job arrivals occur independent of the rate of job completion at the server side. S-client is one such architecture, which defines a scalable method for generating high requests rates that can put the servers in overloaded condition. The S-Client architecture tries to saturate the web server by generating a large number of periodic requests by using low TCP timeout times in order not to be in lock-step with the server.

The original version of S-Client generates a constant request rate and does not provide a mechanism for controlled generation of bursty traffic based on some input parameters. In order to study the effect of collaboration in bursty traffic condition, we modified the original S-Client implementation to generate bursty traffic based on the model formulated by [12]. [12]characterizes the burstiness in request traffic using two parameters, (a, b). Parameter a is the ratio between the maximum observed request rate and the average request rate during the monitoring period. Parameter b is the fraction of time during which the instantaneous arrival rate exceeds the average arrival rate.

A Web server just sees a train of requests, with some inter-request time between two consecutive requests. The characteristics of the request arrival rate can be changed if we manipulate the distribution of the inter-request times. Figure 8 illustrates this concept. Thus, in order to model burstiness, we need to incorporate the two parameters mentioned by [12] into our request generation mechanism.

Suppose that we need to send a total of N requests to the server, with an average request rate r and with burstiness parameters (a, b) as described above. This gives us the total time T for which the request generation mechanism should run:

$$T = \frac{N}{r}$$

Since *a* is the ratio between the maximum observed burst rate *r*+ and the average rate *r*, we know the following relationship exists:

$$a = \frac{r+}{r}$$

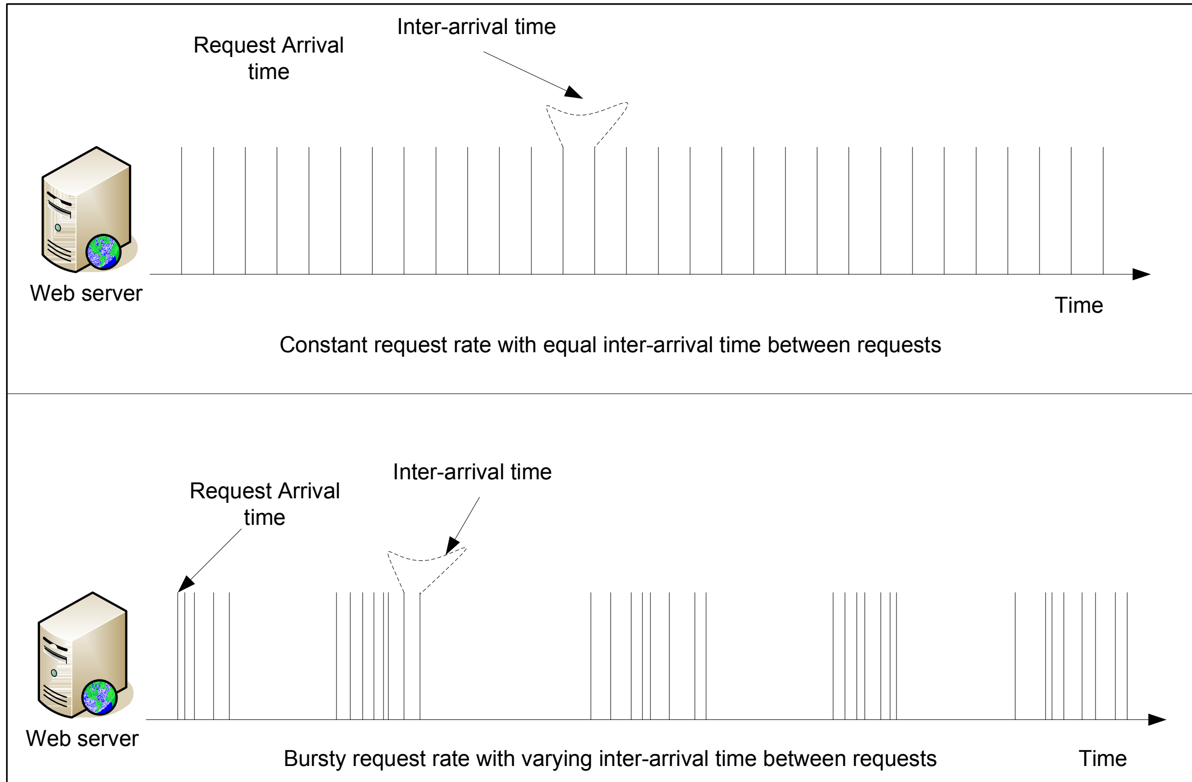


Figure 8 Web servers see a train of requests; the inter-arrival times between requests can be used to characterize their burstiness.

Now, we can define the following parameters:

(i) Arr+ is the total number of requests that should arrive at a rate larger than the average rate and,

(ii) Arr− is the total number of requests that arrive at a rate that does not exceed the average arrival rate

With the definitions of Arr+ and Arr− above, we have the following relationship,

$$N = \mathrm{Arr}{+} + \mathrm{Arr}{-}$$

Also, since b is the fraction of time during which the instantaneous rate exceeds the average rate, $b * T$ is the total time when the instantaneous rate should be r+, thus we have:

$$r+ = \frac{\mathrm{Arr}+}{(b*T)}$$

we know that , $r = \frac{N}{T}$, thus we have:

$$a = \frac{r+}{r} = \frac{Arr+}{(b*T)} * \frac{T}{N} = \frac{Arr+}{(b*N)}$$

thus we have,

$$Arr+ = a * b * N$$

Also, from the relationship between Arr- and Arr+, we can obtain:

$$Arr- = N - Arr+$$

Thus, given the total number of requests, and average request arrival rate and the burstiness parameters, we now have a mechanism to calculate the number of requests that should arrive at an average r (and thus, with inter-request arrival time 1/r) and the number of requests that should arrive at rate r+ (and thus, with inter-request arrival time 1/r+). Further, in order to make the request generation mechanism efficient, we divide the total time for T into n equal sub intervals of duration T/n each called epochs.

Finally, using the definition of parameter b, we can define two important parameters:

(i) the number of epochs in which the request arrival is r+ (high) is:

$$Epoch+ = b * n,$$

and each such epoch should have $\frac{Arr+}{Epoch+}$ requests each with inter-request arrival time $\frac{1}{r+}$.

(ii) the number of epochs in which the request arrival rate is r (average) is:

$$Epoch- = (1-b) * n,$$

and each such epoch should have a total of $\frac{Arr-}{Epoch-}$ requests each with inter-request arrival time $\frac{1}{r-}$.

Now, by interspersing Epoch+ and Epoch− epochs in our request generation mechanism we can produce the required characteristics of average rate and burstiness as required.

5.3 Results

We implemented the core of the architecture as described in Section 4 previously. This implementation is specifically built with the emphasis on measuring the effectiveness of the proposed solution, and is not complete in terms of real deployment of the framework. Aspects such as joining the system, obtaining new neighbors after boycotting a peer server, membership management, neighborhood construction and random checks were left out in the interest of evaluating the performance of the core algorithm.

5.3.1 Single server performance with and without collaboration

In order to evaluate the effectiveness of the collaboration framework, we first need a reference server and measure its performance when it fields client requests in as a standalone server. We used a single Apache 2.2 server as our reference, and used 8 S-Clients (spread across 2 dual processor machines) to generate load on it. The S-Clients generate the HTTP requests at various levels of burstiness. Also, each client sends an HTTP request for a single file of size 15360 bytes, receives the response and then repeats the cycle. Burstiness in the request rate is generated by manipulating the inter-request times using the mechanism explained in Section 5.2. Each S-Client had the connection establishment time set to 500ms. The S-Clients measure the latency of the request by measuring the time between the start of the request, and the time the request is satisfied. Figure 9 plots the average client-perceived latency versus the total connection rate for three different levels of burstiness. The first parameter in the label of each curve is the a factor and the second parameter is the b factor expressed as a percentage. For example, (6, 15) refers to the case in which 15% of the time the request rate is 6 times the average rate.

As we can see from Figure 9, even a small amount of burstiness in the request rate has the capacity to degrade the service provided by the server. For request rates that are not bursty the client side latency is almost linear and the server has no problem in serving the clients. However, as the request rate turns bursty the server starts getting saturated at lower and lower average request rates and the client-side latency increases tremendously. This effect becomes more pronounced for higher average request rates with burstiness factors of (6, 15) and (6, 30).

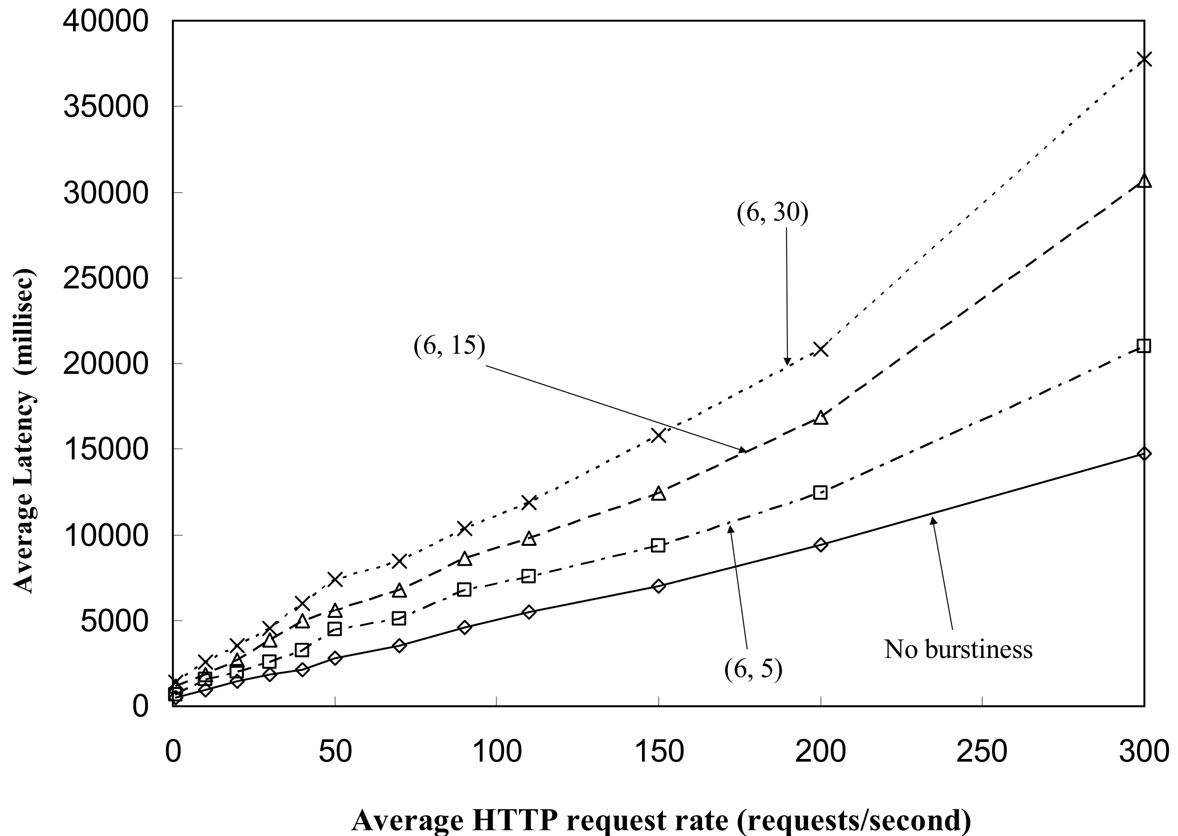


Figure 9 Performance of a standalone HTTP server without collaboration

Further, in order to complete the basic reference, we also measured the performance of this web server when acting in collaboration with an altruistic peer server (a peer who does not free ride and cooperates voluntarily). The peer server uses the same Apache-based configuration. The client set up for the peer server is similar to the reference server except that all the 8 clients generate HTTP requests continuously at 50 requests/sec with burstiness parameters (2, 5). Both servers redirect traffic to each other in a round-robin manner; every alternate request is redirected to the other server. Figure 10 plots the client-perceived latency versus the total connection rate of the reference server when in collaboration with its altruistic peer.

From the results we can conclude that the collaboration with the peer server brings a significant improvement in the reference server's performance. Even for request traffic without burstiness we see that at higher request rates there is an improvement in performance since the benefits of collaboration outweighs the cost of such collaboration (the peer server

routes traffic at ~25 requests/sec with burstiness (2, 5)). The benefits of collaboration are more pronounced for request traffic with high burstiness parameters (6, 15) and (6, 30) where the differences in latencies with and without collaboration are much larger. Further, we also notice that with collaboration the performance of the reference server at low burstiness (6, 5) is comparable to the performance of the server without any burstiness in the request traffic. This shows that collaboration is effective to reduce the performance impact of traffic bursts.

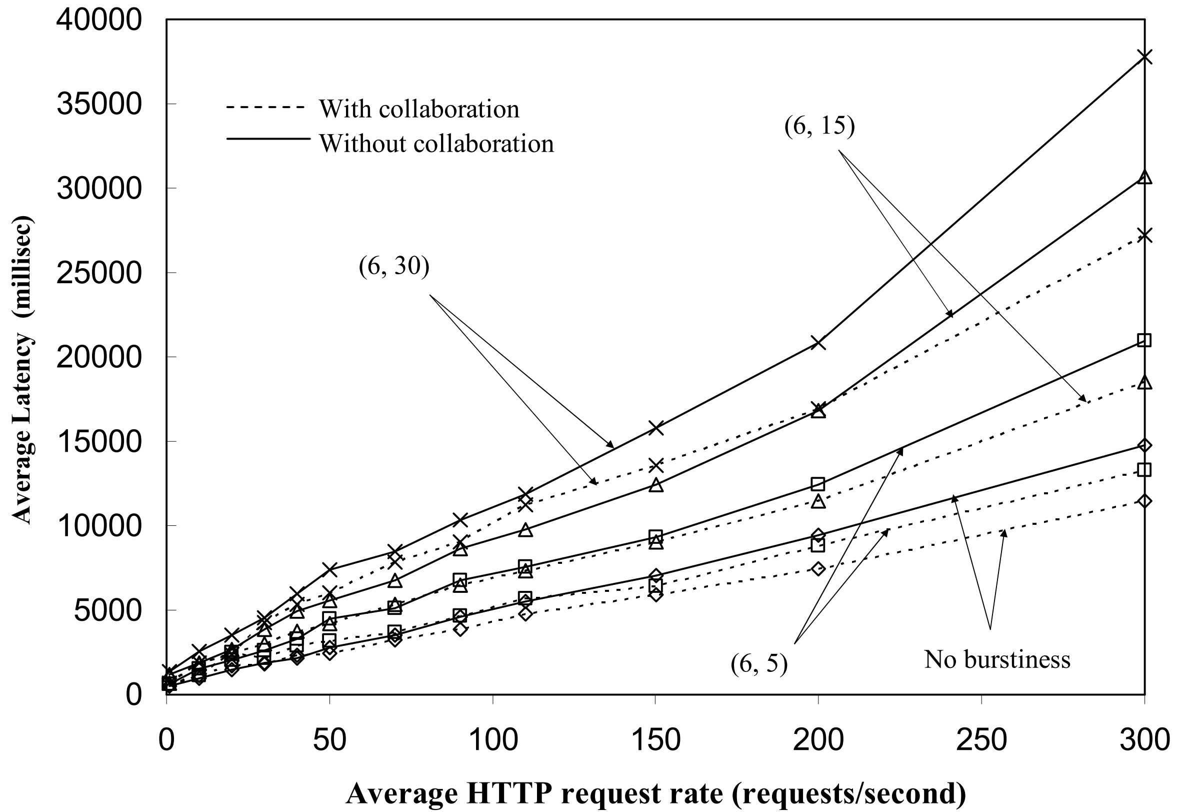


Figure 10 Performance of an HTTP server in collaboration with a peer server

5.3.2 Server performance with free riding

In order to evaluate the advantages of our collaboration system we first examine the effects of free riding on the performance on a single server. To do so, we use the set up described in the section above. However, this time the peer server does not voluntarily co-operate but tries to free ride the system by refusing to honor requests from clients that belong to the original reference server. We define the intensity of free riding as the probability with which the peer will reject a foreign client. Rejected requests are retried back by the client at the origin

server. The origin server does not distinguish between fresh requests and the old request that are retried because the peer server refused to honor them. Figure 11 and Figure 12 show the performance of the original server with the peer server free riding at different intensity levels under traffic with different burstiness parameters.

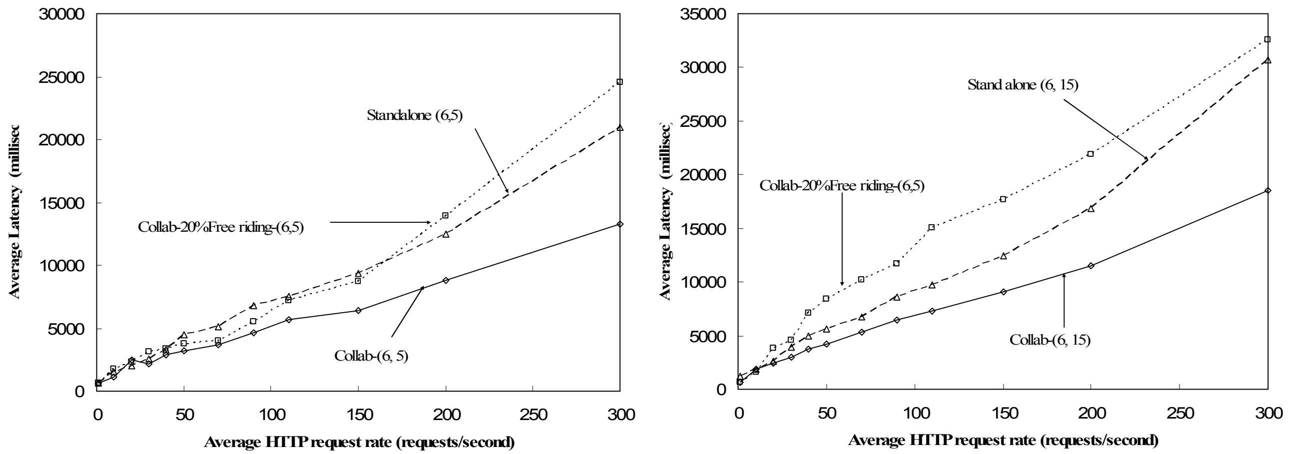


Figure 11 HTTP server performance in collaboration with a peer with 20% free riding level

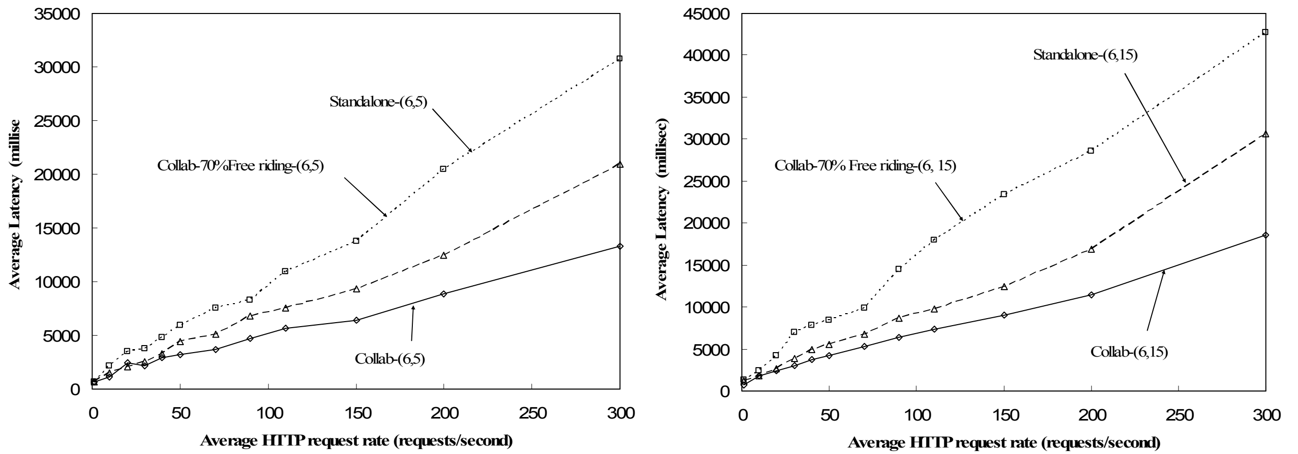


Figure 12 Performance of an HTTP server in collaboration with a peer with 70% free riding level

The most obvious result that can be obtained from the figures above is that a free-riding peer has an adverse effect of the performance of the reference server as compared to the standalone configuration. The performance of the server degrades more as the free riding level of the peer is increased. With 20% free riding level and burstiness parameter of (6, 5) we see that the performance of the server is not tremendously affected by the free riding at low request rates. As the request rate is increased the performance degrades, yet the latencies

are not worse than those obtained when the reference server was in standalone configuration. However, further increase in the request rate brings the performance of the server worse than the standalone case. With 70% free riding level and burstiness parameter of (6, 5) we can see that the performance of the server is comparable to the standalone case at very low request rates but degrades quickly as the request rate increases. From this we can conclude that free riding is much more expensive to the server at higher request rate. In the majority of the scenarios with free riding the origin server is better off without collaboration. This occurs primarily because the bursts are 6 times the average request rate. At higher request rates a burst has a large impact on the performance of the server. Thus, the cost of collaboration (handling foreign clients), and the extra cost of handling redirections (clients who were refused service by the free riding peer) are larger at high requests.

Furthermore, the amount of burstiness has a varying impact on the server performance. With the free-riding level remaining constant, lower amounts of burstiness cause a comparably lower amount of degradation in the performance of the server. As we can see in Figure 11 with a free-riding level of 20% the client perceived latency is much larger for burstiness parameters of (6, 15), than for (6, 5). From this we can conclude that the cost of free riding is higher as the request traffic become more and more bursty. This occurs because with an increase in the burstiness parameter of the request rate, the amount of load on the origin server increases: With (6, 15) burstiness parameter, bursts now occur 15% of the time as opposed to 5% of the time that they would occur with parameters of (6, 5). Due to this increase in load, the origin server is more susceptible to the costs of the free-riding behavior of its peer. Free riding puts the additional load of handling the peer's redirected clients without any reciprocal benefits on an already loaded server. Thus, this additional load is much more expensive to bear as the burstiness of the request rate (and hence the load on the server) increases and we see a drop in performance of the origin server.

In a corresponding manner, Figure 13 shows the average client latency seen by the free-riding peer. As can been seen from the graph, without the enforcement mechanism in place, the free rider sees client latencies lower than that of an altruistic server that collaborates voluntarily. This means that a free rider does better than an altruistic server in an unchecked environment. This is thus an incentive for the nodes to free ride as they can exploit the system to their benefit without contributing their resources in turn.

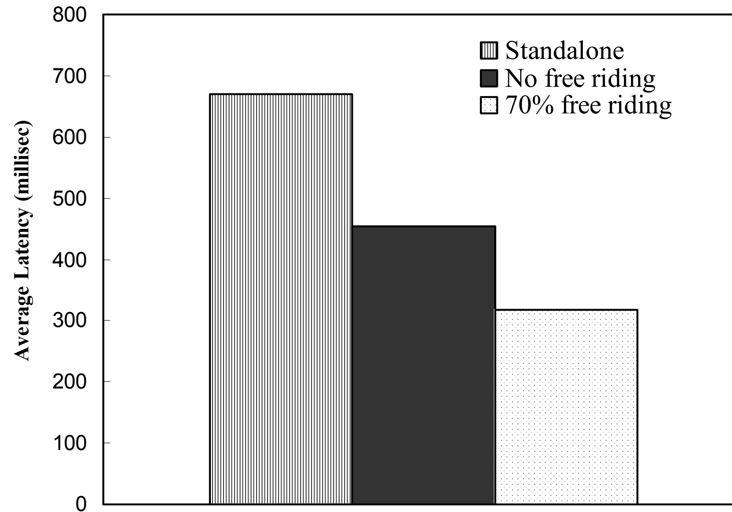
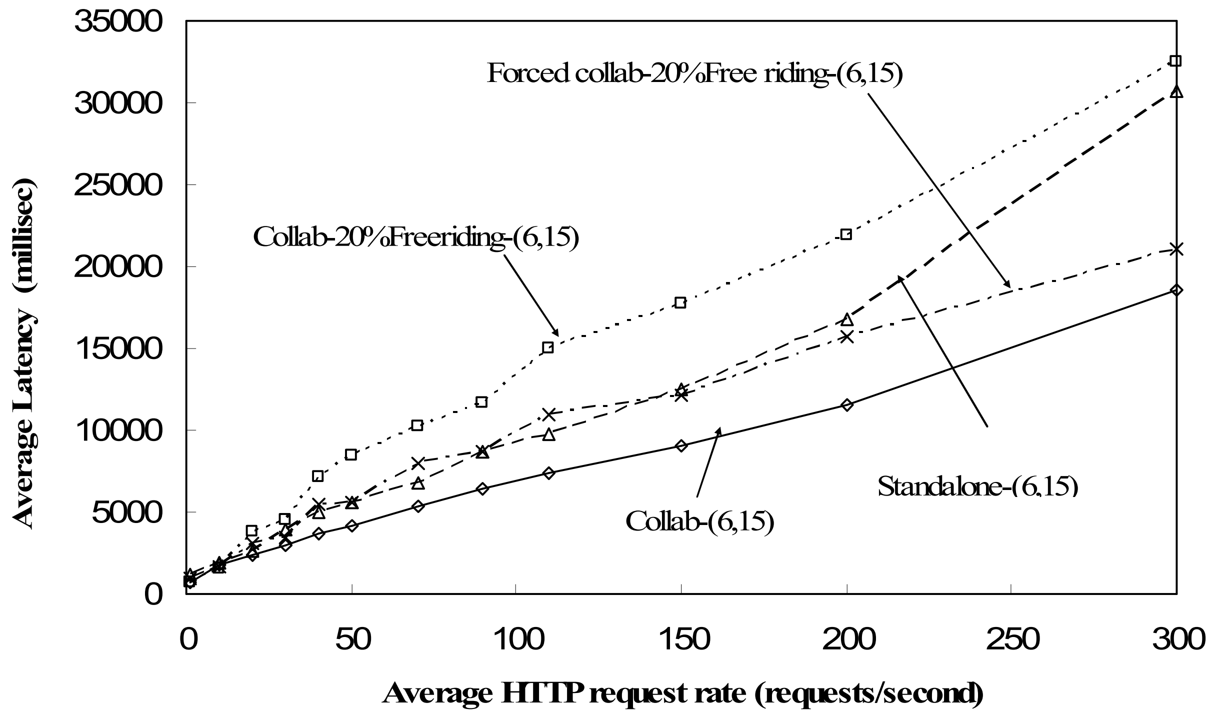


Figure 13 Performance of a free riding HTTP server at 70% free riding level

5.3.3 Server performance with free riding under enforced collaboration

Finally, we conducted a similar set of experiments as the Section above except now the servers are part of collaboration framework developed in this work. These experiments were carried out with an aging parameter of 0.75 for the trends calculations and rate limit factor of 0.5. Initially a debit limit of 30000 bytes is assigned between peers, with an increase parameter of 10. This implies that the deficit limit is increased by 3000 every 30000 bytes contributed by the peer. Further, we use a decrease parameter of 0.1, which implies that the deficit limit is decreased by a factor 0.1 when bad behavior is detected. Again, the peer tries to free ride at varying levels by refusing to honor requests from clients that belong to the reference server. Figure 14 plots the performance of the reference server in this scenario.

From the results we can see that on the whole the free-riding peer is unable to exact as much degradation in the performance of the reference server with our collaboration framework in place as was possible without it. As the figure shows, with the peer free riding 20% of the times, the performance of the reference server follows the standalone case for low request rates and increases further as the requests rate increases. This increase in performance can be attributed to the fact that even with the peer free riding 20% of the time, with the collaboration framework in place, the reference server is able to avail the advantages of the collaboration when the peer in not free riding (which is about 80% of the time).



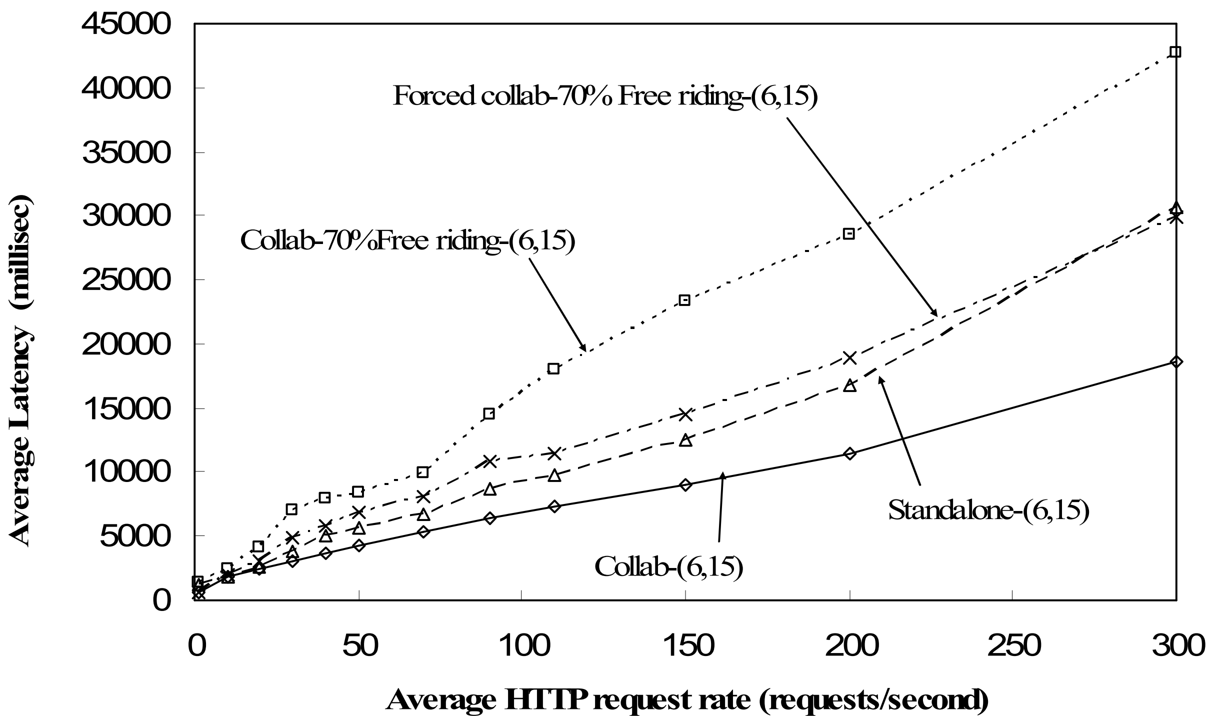


Figure 14 Performance of an HTTP server with free riding peer and forced collaboration

Further, with free riding levels increased to 70% we see that the performance of the reference server follows the standalone case more or less. However, these data illustrate the effectiveness of the framework in that the reference server is almost isolated from the peer and the free riding does not have a negative impact on its performance. At this point there is no advantage of such collaboration and the peer should be boycotted.

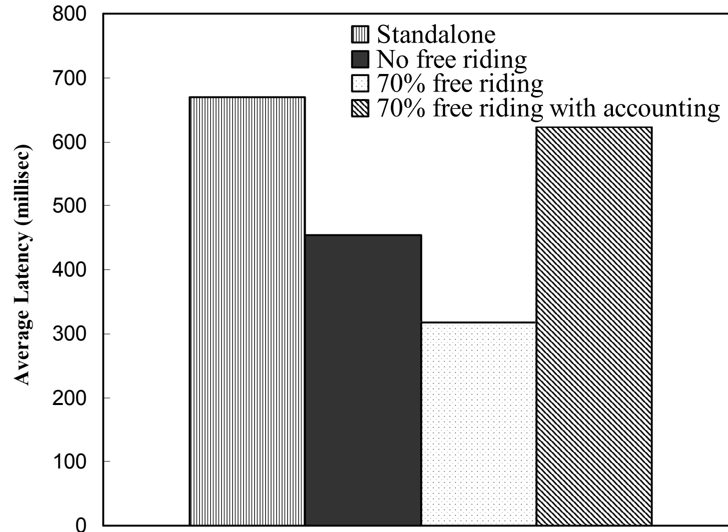


Figure 15 Performance of a free riding HTTP server at 70% free riding level

Figure 15 shows the average client latency seen by the free-riding peer when collaboration is enforced. As mentioned before, without the enforcement mechanism in place, the free rider sees client latencies lower than that of an altruistic server that collaborates voluntarily. However, with 70% free ride level and with collaboration enforced, the average latency seen by the free rider's clients is almost the same as those obtained when it was acting in standalone mode. This confirms that the advantage that free riding can provide is removed when collaboration is enforced. With our mechanism in place, the original server keeps track of the contribution made by the misbehaving node and contributes only in proportion to what it has obtained from the free riding peer. Since the misbehaving node free rides 70% of the time, the original server does not collaborate most of the time and hence we see that the latencies observed by the free rider are similar to its standalone mode. Thus, the incentive to free ride has been removed and the only way the free rider can utilize the advantages of the collaboration is by contributing its own resources to the system.

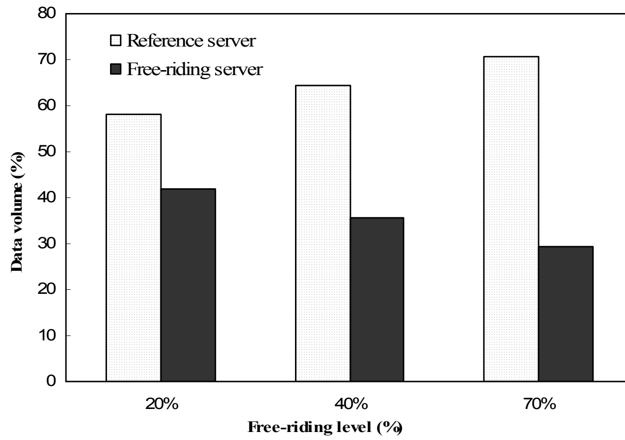


Figure 16(a) Percentage of surrogate data uploaded by the servers, without enforcement.

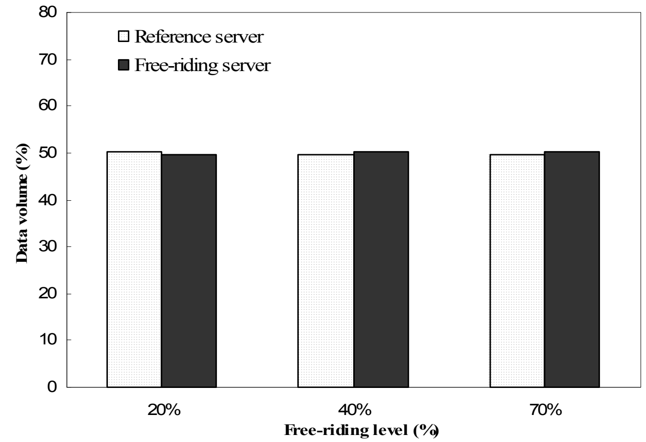


Figure 17(b) Percentage of surrogate data uploaded by the servers, with enforcement.

Figure 16 and Figure 17 show the comparison between the percentage of surrogate data (data served out to foreign clients) served by the servers for a fixed amount of time. Both the servers dealt with a request traffic of 10000 requests at 10 requests/sec with burstiness parameter (6,5), with each request for a file of 5120 bytes. The cumulative data served and consumed by each server was recorded with the peer server free riding at 20%, 40% and 70% of the times without our collaboration framework in place. This experiment was then repeated with the exact setting but with the servers now part of the collaboration framework.

As we can see, obviously the surrogate data served by the reference server is much larger than the data served by the free-riding peer when the collaboration is not enforced. This disparity in contribution becomes larger as the free-riding level is increased. However, with the collaboration mechanism in place, we see that the surrogate data handled by each server is almost the same, i.e. the tit-for-tat strategy ensures that the reference server does not contribute towards the clients of the peer server unless the peer makes a reciprocating contribution itself.

6 Conclusions

CCDNs are designed to share the bandwidth resource of participating nodes for the mutual benefit good of all nodes in the system. By being part of such collaborations, nodes are able to provide good QoS to their clients even in the face of temporary bursts in web traffic. This is done by offloading requests to peer nodes during high request load bursts. However, like any P2P system, CCDNs suffer from the problem of free riders. In the absence of proper incentives the participating nodes will act rationally and may not contribute resources to the system if they can utilize the system's resources for free. Such free-riding behavior can be detrimental to the overall performance of the collaboration.

In this thesis, we proposed a scalable and distributed mechanism for enforcing collaboration among nodes in a CCDN based on DAMD. The system requires that the nodes are divided in to neighborhoods and monitor each other's service parameters during repeated interactions. Nodes keep an account of the data volume and rate contribution made by their neighbors, and indulge in a pair-wise trading scheme using a tit-for-tat strategy. Furthermore, in CCDN environment nodes are not immediate consumers of each other's services, hence contributions made by peers need to be verified using external means. We designed a mobile IPv6 random check mechanism that can be used to verify a peer node's contribution.

Through our experimental evaluation, we have shown that nodes benefit from collaborating with each other. Furthermore, the benefits of collaboration increases as the web request traffic become more and more bursty. However, these advantages are quickly lost as peers in the network start free riding. We have shown that our protocol removes the incentive among nodes to free ride, in that a node is provided service from the system only in proportion to its contribution to its peers. Further, we have also shown that our system prevents the free riders from exploiting good nodes in the system. The performance of a well-behaved node is not affected adversely in the presence of free riders.

Potential areas of future research include the study of system parameters on long term interactions between nodes and to be able to study the behavior of the system in larger deployments. Going further we would also like to investigate the effectiveness of implementing transitive trade between the nodes in the system.

References

[1] E. Adar and B. A. Huberman. “Free riding on Gnutella”. First Monday, October 2000. http://www.firstmonday.dk/issues/issue5_10/adar/index.html. 2000.

[2] G. Hardin. “The Tragedy of the Commons”. Science. 1968.

[3] R. Axelrod. “The Evolution of Cooperation”. NewYork: Basic Books. 1984.

[4] M. F. Arlitt, C. L. Williamson. “Web Server Workload Characterization: The Search for Invariants,” Measurement and Modeling of Computer Systems. 1996.

[5] G. Banga, P. Druschel. “Measuring the Capacity of a Web Server”. USENIX Symposium on Internet Technologies and Systems. 1997.

[6] M. E. Crovella, A. Bestavros. “Explaining World Wide Web Traffic Self-Similarity”. ACM SIGMETRICS 96-5, Philadelphia, PA, USA. 1996

[7] M. Zukerman. “Traffic Modelling and Related Queuing Problems”. Presented at EE Dept., City University of Hong Kong, April, 2002.

[8] J. R. Douceur. “The Sybil Attack”. In First International Workshop on Peer-to-Peer Systems (IPTPS ’02). 2002.

[9] P. Garbacki, A. Iosup, D. Epema, M. van Steen. “2Fast: Collaborative Downloads in P2P Networks”. In Peer-to-Peer Computing, 2006. P2P 2006. Sixth IEEE International Conference on. 2006

[10] G. Pierre, M. van Steen. “Globule: A Collaborative Content Delivery Network”. IEEE Communications, Vol. 44, No. 8. 2006.

[11] S. Jun, M. Ahamad. “Incentives in BitTorrent Induce Free Riding”. In Proceedings of 3rd ACM SIGCOMM Workshop on Economics of Peer-to-Peer Systems (P2PECON). 2005.

[12] D. Menascé, V. Almeida. “Capacity Planning for Web Services”. Prentice hall. 2001.

[13] D. Mosberger, T. Jin, “httperf A Tool for Measuring Web Server Performance”. First Workshop on Internet Server Performance. 1998.

[14] P. Nain. “Impact of bursty traffic on queues”. Statistical Inference for Stochastic Processes, Vol. 5, pp. 307-320. 2002.

[15] J. Shneidman, D. C. Parkes. “Rationality and Self-Interest in Peer to Peer Networks”. IPTPS 2003 : international workshop on peer-to-peer systems N°2, Berkeley CA. 2003.

[16] M. Szymaniak, G. Pierre, M. Simons-Nikolova, M. van Steen. “Enabling Service Adaptability with Versatile Anycast”. Accepted for publication, Concurrency and Computation: Practice and Experience, 2007.

[17] A. Totok, V. Karamcheti. “Improving Performance of Internet Services Through Reward-Driven Request Prioritization”. In Proceedings of the 14th IEEE International Workshop on Quality of Service (IWQoS'06), 2006.

[18] Hierarchical Token Bucket, http://luxik.cdi.cz/~devik/qos/htb/

[19] W. Almesberger. “TCP Connection Passing”. Ottawa Linux Symposium. 2004

[20] http://wiki.coralcdn.org/wiki.php?n=Main.FAQ#runnode

[21] http://www.coralcdn.org/

[22] http://en.wikipedia.org/wiki/Prisoner's_dilemma