HELP:// Hypertext in-Emergency Leveraging Protocol†

MINA GUIRGUIS
Department of Computer Science
Texas State University-San Marcos
msg@txstate.edu

HIDEO GOTO
Department of Computer Science
Texas State University-San Marcos
hg14@txstate.edu

***Abstract*—This paper proposes HELP://, a simple light-weight protocol that runs over HTTP and is used to disseminate information from a server(s) to its clients during the time of a crisis. HELP runs on an architecture that is a hybrid mix between a pure client/server architecture and a Peer-to-Peer architecture. Its resemblance to one versus the other is dynamically decided based on load. In particular, under light load, HELP operates in a client/server mode, where all clients are served directly from the server. Under high load, however, HELP picks one client in every n clients to help the server in serving its content. The value of n is chosen dynamically to optimize the performance of HELP and to ensure that clients receive their requested content with a very high probability, even in the presence of uncooperative clients. We assess the performance of HELP through analysis, simulation experiments and real implementation in Linux. We envision HELP to be installed as a plug-in in common browsers.**

I. INTRODUCTION

Motivation and Scope: Over the past few years, the Internet has evolved into a main source of information that is quite comparable to other sources such as radio and television. Playing this role, however, continuously brings new challenges. In particular, if crises happen – and they do happen – one would expect that many people would turn to the Internet seeking information. This behavior, in a time of a crisis, causes Internet servers to get overloaded due to the high traffic volume from clients, as with the widely cited CNN incident during the September 11th terrorists' attacks, where CNN was not able to respond to the majority of requests for 3 hours [1]. Nowadays, targeting servers running critical activities has emerged into an act of war to cripple countries. For example, several Georgian web sites were subjected to Internet attacks, even before Georgia and Russia entered into a physical war. Moreover, these attacks seemed to be traced back to individuals, who were virtually participating in the war [2]. During such times, it is important to ensure that servers can still perform critical activities and clients can get information about the current incidents.

There has been several approaches to mitigate the overload problem. One approach is to over provision a server to deal with the highest demand. Clearly, this comes at a high cost and a high degree of under utilization during normal operation. A second approach is to use Content Distribution Networks (CDNs). CDNs are typically deployed close to clients to ensure high availability with lower latencies. CDNs, however, are expensive and they do require maintenance. A third approach relies on utilizing an admission controller, whereby requests that would push the server into overload will be rejected [3]–[5]. A fourth approach relies on utilizing a content adaptation controller, whereby the server degrades the content served to cater for a larger subset of clients [6]. Clearly, in the third approach, some clients do not get any access to the information whereas in the fourth approach they receive degraded content. In a time of a crisis, *all clients should be able to access full information about the current incident.*

Currently, the Internet lacks a simple application-level protocol that is able to continue disseminating information in a time of a crisis. The Hyper Text Transfer Protocol (HTTP) and its variants (e.g., HTTPS) are based on a pure "Client/Server" architecture which does not actively seek to mitigate overload. This paper develops a new protocol called "Hypertext in-Emergency Leveraging Protocol" (HELP) that runs on top of HTTP. The main idea behind HELP is to enable a controlled subset of clients to actively participate in the process of delivering content, by acting as intermediate servers for other clients in order to reduce the load on the server. The extent of delegation is directly related to the extent of overload experienced by the server. It is expected that in a time of crisis, clients should cooperate (and help the server) to disseminate information as quickly as possible among them.

In our exposition of HELP, we limit our study to servers serving static content (although it can change over time). Serving dynamic content is harder to achieve since it requires the servers to identify/authenticate each client and to dynamically construct the response. This typically increases the strain on the servers with less room for opportunities to help them.

Paper Organization: In section II, we describe the overall design of HELP, with its goals and components. In Section III, we present simulation and implementation results from our evaluation of HELP. We present a taxonomy of related work in Section IV. We conclude the paper in Section V.

II. THE HELP ARCHITECTURE

HELP is a protocol that combines the advantages of a "Client/Server" protocol and a "Peer-to-Peer" protocol. Its resemblance to one of them versus the other is decided based

† This work was supported by an REP grant from Texas State University.

on load. Under a light load, HELP behaves as a traditional "Client/Server" protocol where the server responds to all clients. As the load on the server increases, HELP delegates more requests to more clients making it behave more like a "Peer-to-Peer" protocol. We illustrate its main operation with an example below.

Clients submit requests to the server using the "help://" protocol requesting a particular file. For simplicity, let's focus on a single file X. Each client includes a secret key K in its request and an open port P. Under light load, the server will respond to every client with the requested file. Once the server detects overload, it would only respond fully to the nth client, where the value of n is dynamically adjusted based on the load. Information (e.g., IP addresses and ports) about the first $n-1$ clients will be recorded, along with their secret keys. The server will respond to the first $n-1$ clients with a `DELEGATED_RESPONSE` message, along with a hash value of the file X. The server will then respond to the nth client with a `SERVING_RESPONSE` message, along with the requested file, the list of clients and their associated keys. The nth client will be the one responsible to serve this file for those $n-1$ clients. Keys are used to prevent a client from being tricked into accepting a file from another entity that is not related to the server. The hash value of the file will ensure that the client serving the file did not modify the file or send an outdated file. Figure 1 illustrates the basic operation of HELP.

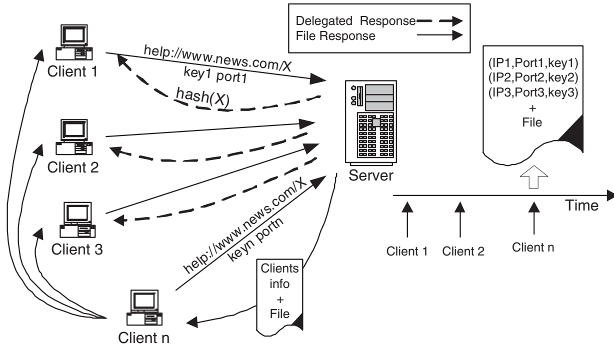

Fig. 1. Basic HELP operation.

A. HELP Design Gaols

We would like to achieve the following goals:

1) *Simple and light-weight.* HELP should be a simple protocol in its use (similar to clients issuing HTTP:// requests from their browsers) and should not require a special Peer-to-Peer network with all of its maintenance operations and management.
2) *Controllability and Accountability.* Servers should have control over their content and its lifetime in the network. Moreover, servers should be able to account for all clients and their downloads.
3) *Content availability.* With a very high probability, all clients should be able to get their requested content even in the presence of uncooperative/malicious clients.
4) *Content authenticity.* All clients should receive authentic content as if they contacted the server directly. A client should never be able to send another client neither modified, nor outdated content.

B. System Components

We model the architecture of HELP with three components, the *server*, the *feedback monitor* and the *delegation controller*.

The Server: The server responds to clients' requests. In HELP, there are three kinds of responses:

Case (1): The server responds directly to a client with the requested file. This case happens under light load, where HELP acts as a traditional client/server architecture. We refer to this client as a *direct client*.

Case (2): The server responds directly to a client with the requested file, along with a list of other clients requesting the same file. This list includes the clients' IP addresses, ports and keys. The client receiving this information will serve the file to those clients. We refer to this client as a *serving client*.

Case (3): The server responds with a delegation expected along with a hash value of the requested file. The client will not receive the file directly from the server, but from another client. We refer to this client as a *delegated client*.

The Feedback Monitor: The feedback monitor periodically measures the server's utilization and reports the value back to the delegation controller.

The Delegation Controller: The delegation controller decides the value n, where 1 client in every n clients would receive direct content from the server and would be asked to serve the other $n-1$ clients. This client is chosen uniformly at random among the n clients. In this paper, we use a Proportional Integral (PI) controller [7] to adjust $n(.)$ based on the deviation of the server's utilization $\rho(.)$ from a desirable set value ρ^*. At time instant i, the controller adjusts the value $n(i)$ based on the following equation:

$$n(i) \quad = \quad n(i-1) + G \times (\rho(i) - \rho^*) \qquad (1)$$

where G is the gain of the system that determines the aggressiveness of the controller in reaction to the difference between $\rho(i)$ and ρ^*. The value of G is typically chosen low enough to guarantee stability yet high enough to ensure responsiveness. In our experiments, we used values within this range.

Notice that with our uniformly random choice of selecting 1 client in every n clients, we may be at the risk of waiting for a long period of time to find that one client. Let D be a random variable that denotes the number of clients who get delegated until we find that one client to serve. If we simply choose clients with a probability of $\frac{1}{n}$, then D is a geometric random variable and the probability of waiting for $n-1$ clients before choosing the serving client is:

$$Prob[D = n] \quad = \quad (1 - \frac{1}{n})^{n-1}\frac{1}{n} \qquad (2)$$

In order to guarantee that HELP does not wait for a long time before finding the serving client, we modify the probability of picking the serving client from simply $\frac{1}{n}$ to $\frac{1}{n-C}$, where C is the number of clients that got delegated, so far. This choice would cause D to be a uniform random variable and the probability of D going beyond n is 0. This method was used in [8].

C. Security Issues

So far, we have focused on the case where 1 client serves $n-1$ clients. This may raise some problems due to clients going off-line, clients being uncooperative (free riders) or due to the presence of malicious clients. Regardless of the real intent, we will refer to those clients as uncooperative clients, since they will *not* serve the file to the rest of the clients. To ensure that all clients would receive their requested content, the server should send the file (along with the list of clients) to more than just 1 client, for every n clients. Next, we drive the optimal number of clients that should act as servers, to bound the probability of a failure (clients do not receive the requested file, since the serving clients turn out to be uncooperative).

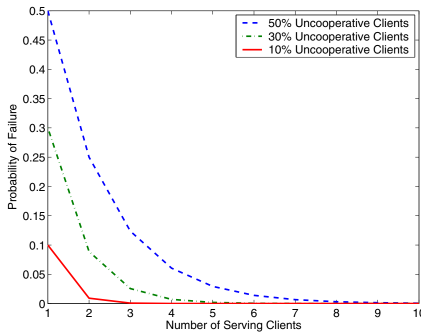

Fig. 2. Probability of failure versus the number of serving clients chosen.

Consider n clients, b of them are uncooperative clients. We denote $Prob[k \rightarrow b]$ as the probability of failure due to the server choosing k clients and all of them turn out to be uncooperative.[1] The case with serving 1 client in every n clients, leads to a failure probability, $Prob[1 \rightarrow b]$, of $\frac{b}{n}$. The case with serving 2 clients in every n, leads to a failure probability of $\frac{b(b-1)}{n(n-1)}$, since a server should not serve the same client more than once. The general case of serving k clients in every n clients, leads to a failure probability of $Prob[k \rightarrow b]$ that is given by:

$$Prob[k \rightarrow b] \quad = \quad \frac{b}{n} \times \frac{b-1}{n-1} \times \frac{b-2}{n-2} ... \times \frac{b-k+1}{n-k+1} \quad (3)$$

Figure 2 illustrates the probability of failure as the number of clients k is varied. We do so for 3 different cases of uncooperative clients. We take n to be 100 clients, and we study the cases where the uncooperative clients are 50%, 30% and 10% of the clients n. Notice that the probability of failure decreases exponentially fast. With 50% uncooperative clients, the server should send the file to 8 clients out of a 100 so that the probability of failure is below 0.3%. To drive the optimal number k, we bound the probability of failure, so that with a very high probability all clients receive their files.

[1] This is a modified version of the balls and bins problem. We have n bins, b of them are bad ones. We throw k balls uniformly at random, and we ask what is the probability that they all fall in bad bins. A bin is used only once.

To ensure data integrity and that clients do not modify a file (willingly or unwillingly), nor send an outdated file, we request the server to respond to clients with a hash value (e.g., MD5 or SHA-1) for the requested file. When a client receives the file from another client, he/she would compare the hash values. If the values match, the file is accepted, otherwise, the file is rejected.

D. Deployment Issues

In this subsection, we report on deployment specifics of HELP. HELP is a text-oriented protocol and uses HTTP [9] at its core. There are two kinds of messages in HELP, request messages and response messages.

```
START_LINE       GET / HTTP/1.1 HELP/1.0
MESSAGE_HEADER   PORT: 3430
                 KEY: 4A3B87C32
                 HASHING_ALG: "MD5"
                 HOST: www.a-news-website.com

START_LINE       PUT / HTTP/1.1 HELP/1.0
MESSAGE_HEADER   KEY: 4A3B87C32
MESSAGE_BODY     Requested_file
```

Fig. 3. HELP: Request messages format; request from a client to a server (top) and request from a serving client to a delegated client (bottom).

Request Messages: In HELP, when a client requests a file from the server, it sends a request message as shown in figure 3 (top). The port number indicates an open port which *may* be used later, if the client receives a DELEGATED_RESPONSE. The key field indicates a pass code chosen by the client which *may* also be used if the client gets a delegation response. The hashing algorithm field indicates which hashing method the server should use when responding with a hash value for the file (e.g., md5 versus sha-1). A direct client, will close its open port once it starts receiving the file. A delegated client will need to keep the port open, since the serving client will be sending the file on that port. If a delegated client does not receive the requested file on its open port within a specific time (e.g., 120 seconds), it will close this open port and retries its request to the server. A serving client receives the requested file in the same exact manner as a direct client, except that it needs to (1) close its own open port and (2) parse the list of clients and respond to each one of them with the file received. Each response will occur on the client's open port and will include the client's key.

Another form of request messages occurs when a serving client is sending a file to a delegated client. The header will include the key. The file itself will be in the message body. Figure 3 (bottom) illustrates this message format.

Response Messages: In HELP, there are three forms of responses from the server: `DIRECT_RESPONSE`, `DELEGATED_RESPONSE`, or `SERVING_RESPONSE`. The format of the messages is given in Figure 4. In the case

of a direct response, the file is included in the message body as shown in Figure 4 (top). In the case of a delegated response, a hash value of the file is returned based on the hashing algorithm requested by the client as shown in Figure 4 (middle). In the case of a serving response, a list of clients is included in the message header and the file in the message body as shown in Figure 4 (bottom).

```
START_LINE       HTTP/1.1 HELP/1.0 202 ACCEPTED DIRECT_RESPONSE
MESSAGE_HEADER   HTTP headers only
MESSAGE_BODY     Requested_file

START_LINE       HTTP/1.1 HELP/1.0 202 ACCEPTED DELEGATED_RESPONSE
MESSAGE_HEADER   HASH: d41d8cd98f00b204e9800998ecf8427e

START_LINE       HTTP/1.1 HELP/1.0 202 ACCEPTED SERVING_RESPONSE
MESSAGE_HEADER   CLIENT_LIST:
                 <IP: 147.26.101.179, PORT: 3430, KEY: 4A3B87C32>
                 < more clients information  >
MESSAGE_BODY     Requested_file
```

Fig. 4. HELP: Response message format; top (direct response), middle (delegation response) and bottom (serving response).

Implementation Details: A server running HELP employs a resource monitor daemon that measures the bottleneck resource utilization periodically. This may be the CPU usage, the memory usage, the network usage, error rate, or the number of pending requests inside the system. Also, one can think of a combination of these metrics. In our experimental implementation of HELP in Section III, we used the network utilization. The monitor daemon communicates the measured utilization with a client-cgi through IPC Message Queue. The client-cgi decides which response a client gets.

The client implementation uses a file-recv daemon that runs on an open port indicated in the HELP request message. In the case of receiving a delegation response, file-recv daemon will be waiting for the file from another client on that port. Notice that HELP can be incrementally deployed via the "User Agent" field to identify if the client is running HELP or not.

III. Experimental Evaluation

In this section, we report on a representative subset (due to space limitation) of our experimental results.

A. Simulation Results

To investigate the effectiveness and performance of HELP, we wrote a discrete-time event simulator. For simplicity, we assume that the server is serving a single file and that the serving time is fixed to 100 msec of processing time. The serving time for a delegated request is 10 msec. Clients submit requests to the server at a rate λ requests per second. We vary λ to simulate overload, and we observe the queue size, the response time observed by the clients and percentages of delegations. We fix G to 0.1 and we choose the queue occupancy as our measure of utilization. The target utilization is fixed to 10. Each experiment concludes after 100,000 requests get processed by the server.

Figure 5 (right) shows the queue size and the average response time for different values to λ. Notice that for values of λ larger than 10, the system is already in overload conditions, since the arrival rate is higher than the departure rate (which is 10 requests per second, since the processing time takes 100 msec). One can observe that the queue size is well maintained below its target and approaches the target as the load increases. We plot two lines for the average response time; one for the requests that were processed by the server (direct and serving) and one for the total requests. One can observe that requests served by the server maintain a bounded response time. The total average request time includes those that are delegated (their response time is computed based on when the serving client receives the file, plus their waiting time experienced as the serving client responds to all clients).

Figure 5 (middle) illustrates the average number of delegations processed by each serving client and the percentage of the total requests delegated. One can see that the average number of delegations is small (below 15 for an increase of 3 times the arrival rate).

B. Implementation Results

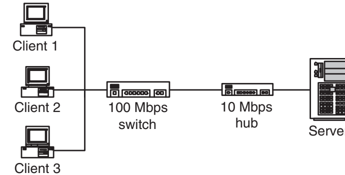

Fig. 6. Experimental setup used in our implementation of HELP.

Experimental Setup: Figure 6 illustrates the experimental setup we used in the implementation of HELP. It is composed of 3 clients and a server. All machines are Xeon 3.2 GHzx1 with 4GB of memory. They all run Redhat Enterprise Linux 4-x86 32 bit. The server runs Apache HTTP server 2.0.52 [10]. The clients use Httperf 0.9.0 [11] to generate requests to the server. We have created a bottleneck on the server's side by constraining the bandwidth to 10 Mbps. The effective bottleneck capacity is 9.3 Mbps (obtained from an experiment). We used the bandwidth utilization as a measure of congestion that would cause delegation of requests among clients.

To generate the workload, each client machine was generating 3 requests per second. Each request requires the server to respond with a 200KB file, which takes about 170 msec (in case of a direct response from the server without delegations). Thus the theoretical limit for the server is about 5.8 requests per second. All clients run HELP and the server delegates requests among them based on equation 1. The target utilization of the bandwidth was fixed to 5 Mbps. Figure 5 (right) illustrates the average response time experienced by all clients over time. We illustrate three cases; without HELP, with HELP and G is chosen to be 0.5 and with HELP and G is chosen to be 1. One can see that without HELP, it takes the server a long time to even finish responding to requests (with a continuous increase in the response time due to accumulating overload). With HELP, the server delegates requests among clients to keep the bandwidth utilization within target while keeping the average response time, for all requests, low.

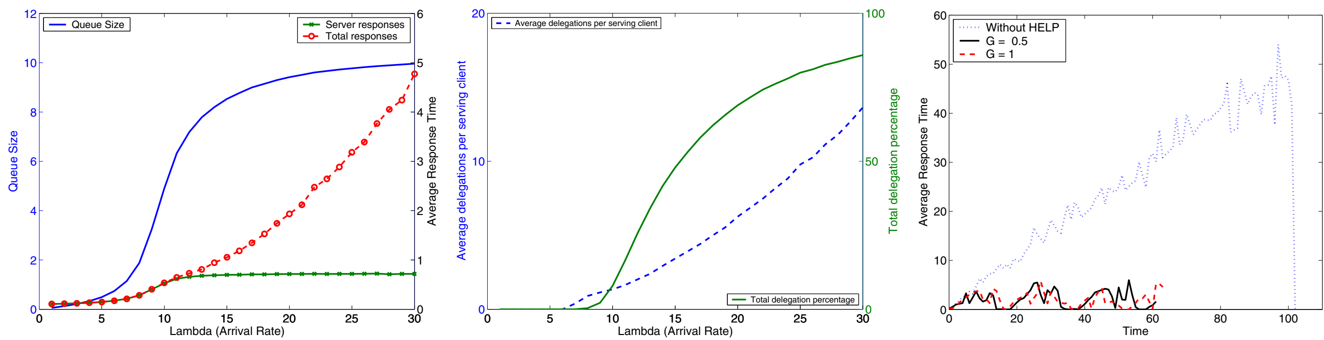

Fig. 5. Experimental evaluation with simulations (left and middle plots) and implementation (right plot).

IV. Related Work

HELP relates to two main areas of research: (1) Server overload and (2) Peer-to-Peer networks.

Server Overload: There is a large body of work that investigated the server overload problem and proposed different approaches to mitigate it. Such approaches can be broadly classified along two dimensions, static and dynamic. Static polices aim to increase the capacity of the system by over-provisioning the resources needed. Dynamic policies, on the other hand, rely on adapting the traffic workload or the operation of the resource, based on the level of load. These polices are typically employed via load-balancers [12], admission controllers [13], [14], schedulers [15] and content adaptation controllers [6], [16]. Load balancers and schedules, do not mitigate overload but aim to improve the response time for clients. Admission controllers and content adaptation controllers fail to respond to all clients with full content.

Peer-to-Peer Networks: There has been some work that used Peer-to-Peer (P2P) networks to handle flash crowds [17], [18] and Denial of Service (DoS) attacks [19], among others. In [17], PROOFS is proposed as a P2P network to handle flash crowds. PROOFS constructs a P2P network where neighbors are selected at random. Queries for objects are flooded (with a particular scope) through the P2P network rather than to the server. In [18], an overlay of cache proxies is formed to handle flash crowds. This method uses DNS redirection to point the clients to a cache proxy rather than to the server. In [19], MOVE was proposed as a P2P network that can route requests to the current location of services (which changes under DoS attacks). HELP, on the other hand, does not need to maintain a P2P network, utilize caches or change DNS entries.

V. Conclusion

In a crisis situation, many people turn to the Internet seeking information. This in return causes a lot of strain on servers as evident from past incidents. In this paper, we have proposed HELP, a simple, light-weight application level protocol that can continue to disseminate information with the help of clients. HELP runs over HTTP and does not need to maintain a separate peer-to-peer network (saving significant overhead for maintenance). Moreover, it does not require changes to DNS entries. HELP can be easily implemented as a plug-in in common browsers. We are currently investigating the exact interface between HTTP and HELP and efficient strategies for forming client swarms.

Acknowledgment

The authors would like to thank Azer Bestavros for his feedback on this work.

References

[1] "Internet Under Crisis Conditions: Learning from September 11," National Research Council, 2003.
[2] "New breed of hackers tracking a new breed of cyberattackers," The Washington Post, Sept, 2008.
[3] M. Welsh and D. Culler, "Adaptive Overload Control for Busy Internet Servers," in *Proceedings of the 4th USENIX Conference on Internet Technologies and Systems (USITS)*, March 2003.
[4] M. Welsh, D. E. Culler, and E. A. Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services," in *Symposium on Operating Systems Principles*, 2001, pp. 230–243.
[5] A. Bestavros, N. Katagai, and J. Londono, "Admission Control and Scheduling for High Performance World Wide Web Servers," Tech. Rep. BUCS-TR-1997-015, Boston University, CS Department, Aug 1997.
[6] T. F. Abdelzaher and N. Bhatti, "Web Content Adaptation to Improve Server Overload Behavior," *Computer Networks*, vol. 31, 1999.
[7] K. Ogata, "Modern Control Engineering, 4th Ed.," Prentice Hall, 2002.
[8] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *Trans. on Networking*, August 1993.
[9] "Hypertext Transfer Protocol," RFC 2616.
[10] "Apache HTTP Server," http://httpd.apache.org.
[11] D. Mosberger and T. Jin, "Httperf: a tool for measuring web server performance," in *Proceedings of the First workshop on Internet Server Performance*, Madison, WI, June 1998.
[12] V. Cardellini, M. Colajanni, and P. Yu, "Dynamic load balancing on web-server systems," *IEEE Internet Computing*, 1999.
[13] M. Andersson, M. Kihl, and A. Robertsson, "Modelling and Design of Admission Control Mechanisms for Web Servers using Non-linear Control Theory," in *Proceedings of ITCom*, September 2003.
[14] A. Robertsson, B. Wittenmark, and M. Kihl, "Analysis and Design of Admission Control Systems in Web-server Systems," in *Proceedings of American Control Conference*, June 2003.
[15] B. Schroeder and M. Harchol-Balter, "Web servers under overload: How scheduling can help," *ACM Transactions on Internet Technology*, 2006.
[16] M. Andersson, M. Host, Jianhua Cao, C. Nyberg, and M. Kihl, "Design and Evaluation of an Overload Control System for Crisis-Related Web Server Systems," in *Proceedings of ICISP*, France, Aug 2006.
[17] A. Stavrou, D. Rubenstein, and Sahu S, "A Lightweight, Robust P2P System to Handle Flash Crowds," *IEEE Selected Areas in Communications*, January 2004.
[18] C. Pan, M. Atajanov, M. Hossain, T. Shimokawa, and N. Yoshida, "FCAN: Flash Crowds Alleviation Network Using Adaptive P2P Overlay of Cache Proxies," *IEICE Trans on Communications*, April 2006.
[19] A. Stavrou, A. Keromytis, J. Nieh, V. Misra, and D. Rubenstein, "MOVE: An End-to-End Solution To Network Denial of Service," in *Proceedings of the SNDSS*, Feb 2005.