A Multicast-On-Large-Demand Approach to the Flash Crowd Problem

Rocky K. C. Chang
Department of Computing
The Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong
Email: csrchang@comp.polyu.edu.hk

ABSTRACT

In this paper we propose a multicast-on-large-demand (MOLD) approach to the flash crowd problem. A MOLD Web server may dynamically open a multicast channel for resources when detecting a very high demand for them, and it reverts back to the normal unicast mode when the flash crowd subsides. A number of mechanisms necessary for realizing the MOLD system are thoroughly discussed. We have implemented the MOLD system in Java and evaluated the performance in a test-bed.

Categories and Subject Descriptors

C.2 [**Computer Systems Organization**]: Computer-Communication Networks; I.6 [**Computing Methodologies**]: Simulation and Modeling

General Terms

Measurement, Performance, Design

Keywords

Flash Crowds, Hypertext Transfer Protocol, Multicast

1. INTRODUCTION

Flash crowd refers to a sudden surge in the demand for certain Web resources in the Internet. Therefore, similar to the effect of a distributed, denial-of service (DDoS) attacks, flash crowd could exhaust a Web site's resources for accepting new (TCP) connections, processing capability, or network bandwidth [1]. The demand unpredictability makes it very difficult to pre-allocate adequate resources to meet the demand surge. Even when the demand is quite predictable, providing sufficient resources just for the flash event is too costly a general solution [2].

2. DESIGN OBJECTIVES

1. HTTP extension: The existing HTTP protocols need extensions to support dynamic opening and closing of multicast channels.
2. HTTP proxying: MOLD proxies must be able to support all possible scenarios involving multicast and unicast channels.
3. On-large-demand multicast channels: An MOLD server must be able to open a multicast channel for very popular resources, in additional to the unicast channels.
4. Multiplexing multiple resources: An MOLD server must be able to aggregate a number of very popular resources in a single multicast channel.
5. Multicast population monitoring: An MOLD server must be able to continuously monitor the population size of a multicast channel in a scalable manner.
6. Adaptive push rate: An MOLD server must be able to adaptively adjust the push rate based on the change rate of the resources sent in the channel.
7. Switching to multicast channels: MOLD clients and proxies must switch from the unicast channel to the announced multicast channel if they support MOLD.
8. Data reliability: The proposed architecture does not assume the availability of forward error correction codes or reliable multicast protocols.
9. Monitoring the channel quality: MOLD clients and proxies must be able to monitor the quality of the multicast channel.
10. Switching back to unicast channels: When switching from a multicast channel back to a unicast channel, the switching must be performed gracefully.

3. ARCHITECTURE AND PROTOCOLS

3.1 New HTTP Headers

We have defined a minimal set of new HTTP headers to support MOLD. To ensure full compatibility and transparency, we propose that the HTTP version is still maintained at 1.1. In Figure 1, we show the possible states for a resource.

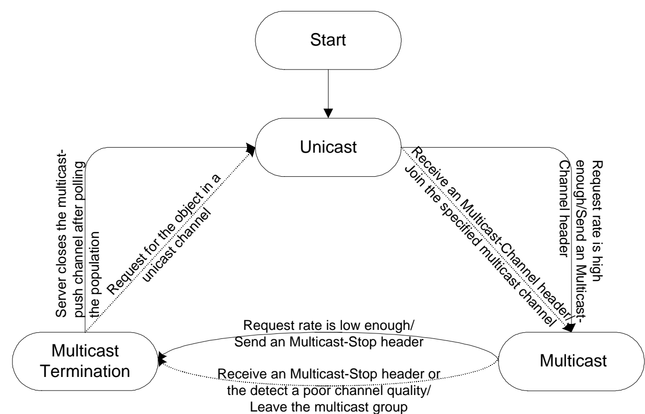

Fig. 1. States for a resource kept by MOLD clients and servers. There are two types of state transitions: the solid ones for servers, and the dotted ones for clients. Regular clients and servers are always in the Unicast state. MOLD proxies behave like servers when they multicast resources to clients, and they behave like clients when they are receiving resources via multicast channels.

3.2 Opening Multicast Push Channels and Join Latency

An MOLD server may use a request-rate list (RR-list) and a simple threshold-based decision rule to determine whether a multicast channel should be open for a resource. The request rate can be computed based on weighted time averages. That is, the request rate at the end of *i*th interval, denoted by R_i, is given by $R_i = \alpha M_i + (1 - \alpha)R_{i-1}$, where M_i is the instantaneous rate obtained during the *i*th interval.

3.3 Estimating Multicast Receiver Population

There are a few important reasons why an MOLD server needs to know the population size. The primary one is for the server to determine when to close the channel because there are not enough receivers to warrant for it. Another one is to control the size of the multicast group. Managing a multicast group becomes more difficult and inefficient when the group size becomes very large.

3.4 Determining Multicast Push Rates

Recall that an M-list contains resources that are delivered in a multicast channel, in addition to the regular unicast channel. An important issue to consider concerns how often these resources are pushed into the channel. One simple mechanism is to push a resource whenever it is changed. This asynchronous mechanism, however, suffers from several problems. As a result, we propose to use cyclic transmission schedules to deliver the resources in the M-list. During each cycle, a resource is pushed out at least once.

3.5 Closing Multicast Channels

When an MOLD server decides to move to the Multicast Termination state, it multicasts Multicast-Stop headers several times to make sure that all receivers have received the message. Moreover, an MOLD server may perform population poll before closing the multicast channel. After receiving 0 replies after a consecutive number of polling, the server may safely assume that no one is tuned to the multicast channel and therefore change to the Unicast state.

3.6 Monitoring the Quality of Multicast Channel

MOLD client and proxies include a simple timeout mechanism to detect possible multicast channel problems. Recall that an MOLD server includes R_A in the Multicast-Channel headers, along with the multicast address and port number. With R_A, an MOLD client can choose the timeout value for a requested resource to be σ/R_A, where σ is least equal to 1. If the requested resource does not arrive within σ/R_A, the channel is not usable.

4. PERFORMANCE STUDIES

Fig. 2 shows the set-up for the performance studies. All the machines are running RedHat Linux 7.3. Since there will be a lot of concurrent TCP connections during the test, the TCP FIN timeout value is reduced from 180 seconds to 10 seconds, and the TCP keep-alive timeout value is reduced from 7,200 seconds to 30 seconds.

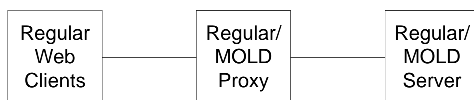

Fig. 2. The experiment setup, consisting of an MOLD server, an MOLD proxy, and a large number of regular Web clients.

Fig. 3 shows that the average response time rises rapidly with request rate in the cases of unicast delivery. The excessive delay is partially due to the busyness of the server in handling a large number of requests. All three MOLD cases, on the other hand, keep the response time very low.

In Fig. 4, all the unicast-pull cases again incur a very large CPU utilization on the server. Although the HTTP/1.1-Pull case incurs the least among the three, they all reach 100% CPU utilization when the request rate reaches 2,500 per minute. The main reason contributing to such high CPU utilizations is due to the fact that the Web server is required to keep a large number of socket connections. There is a protocol control block (PCB) table in the Web server OS that keeps the network connection states. A large PCB table not only consumes more kernel memory, but also consumes more CPU cycles. When the MOLD mechanism is used, the CPU utilizations on Web server do not rise higher than 90% in all the tests, because the server no longer keeps a large number of TCP connections.

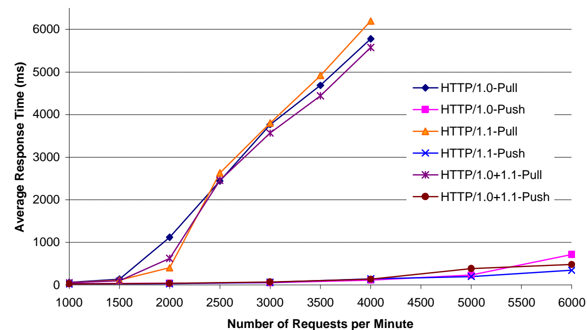

Fig. 3. Average response time vs. request rates.

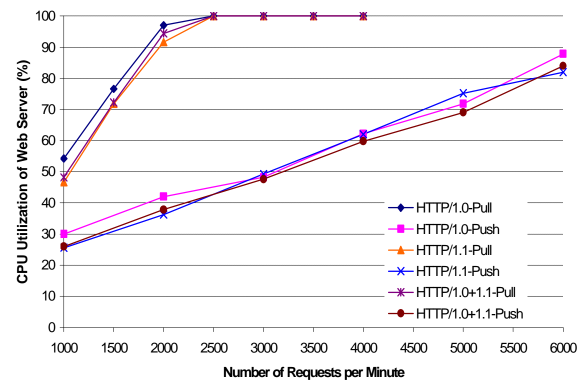

Fig. 4. CPU utilization of the Web server vs. request rates.

5. ACKNOWLEDGMENTS

The work described in this paper was partially supported by a grant from The Hong Kong Polytechnic University (Project No. COMP-H-ZJ83).

6. REFERENCES

[1] R. Chang, "Defending Against Flooding-Based, Distributed Denial-of-Service Attacks: A Tutorial," *IEEE Communications Magazine*, vol. 40, no. 10, 2002.

[2] A. Iyengar, et al, "High-Performance Web Site Design Techniques," *IEEE Internet Computing*, pp. 17-26, Mar/Apr, 2000.