Load-Balanced Agent Activation for Value-added Network Services

Chao Gong [*], Kamil Sarac, Ovidiu Daescu,
Balaji Raghavachari, Raja Jothi

Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083, USA

Abstract

In relation to its growth in size and user population, the Internet faces new challenges that have triggered the proposals of value-added network services, e.g., IP multicast, IP traceback, DiffServ, IntServ, etc. In addition, recent advances in processor and hardware techniques have enabled the production of high speed and powerful routers. Therefore, it is not unreasonable to expect the Internet to provide a variety of value-added network services other than packet forwarding in the near future. Depending on their purposes, value-added services may improve the scalability and efficiency of end user applications or may enhance the reliability and security of the network infrastructure. On the other hand, they may incur non-trivial overhead on the routers providing these services. It is a thorny problem to reach a balance between the performance of value-added services and the incurred overhead. In this paper, we study this problem in the context of both reliable multicast and distributed denial-of-service (DDoS) defense. In either scenario, a software agent is activated at some routers in a tree topology to provide the required functionality. We formulate the problem as load-balanced agent activation problem (LBAAP). Our goal is to develop a mechanism to activate value-added service agents in the network for the purpose of reaching a balance between the performance and overhead. We develop a polynomial time algorithm to solve the LBAAP problem in single tree case, and propose a heuristic for the LBAAP problem in the case where multiple trees exist in the network, a problem we conjecture is NP-hard. Finally we evaluate the performances of various approaches for activating value-added service agents through simulation.

Key words: Load balancing, value-added service, reliable multicast, DDoS defense

[*] Corresponding author. Tel.: +1-972-883-2185; fax: +1-972-883-2349.
Email address: gong@student.utdallas.edu (Chao Gong).

1 Introduction

During the recent years, several new technologies/services have been proposed/introduced into the Internet. These include quality of service (QoS) routing [1,2], content delivery networks [3], multicast communication services [4], and so on. In addition, the increasing rate of denial-of-service (DoS) attacks in the Internet has clearly shown the need for effective DoS defense mechanisms most of which require additional support from the network [5–7].

One common characteristic of these technologies is that they introduce additional functionalities into the network devices, i.e., routers and/or servers co-located with routers. In this paper, we refer to this type of additional functionalities as value-added network services (VAS). VASs are typically implemented as software modules at routers and/or co-located servers. We refer to this type of software modules as VAS agents. A VAS agent at a network device can be turned on (activated) or turned off (deactivated). If a VAS agent is activated at a device, we say that the device hosts the VAS agent. VAS agents help to improve the performance of end user applications and to enhance the robustness of the network infrastructure. However, VAS agents incur non-trivial overhead on the devices hosting them.

In this paper, we use reliable multicast and distributed denial-of-service (DDoS) defense as example services to study the tradeoff between the VAS performance and the overhead introduced by VAS agents on the network devices. Our goal is to develop a mechanism to activate VAS agents on a proper set of network devices so that (1) VAS performance requirements are satisfied and (2) the resulting assignment does not cause any load imbalance among the network devices.

1.1 Reliable Multicast

Reliable multicast is a value-added network service that provides reliable data transport from a single source to multiple receivers in the Internet. A key challenge in reliable multicast is scalability. The main difficulty in making reliable multicast scalable is the feedback message implosion at the multicast source site. As the number of receivers increases, their feedback messages back to the source could eventually overwhelm the computing resources and even the link bandwidth at the source site.

The usual approach to ensure reliable delivery in reliable multicast protocols is the use of Negative Acknowledgements (NAK). That is, receivers send out NAK messages to inform the source about packet loss. Compared with Positive Acknowledgement (ACK), NAK in reliable multicast can alleviate feedback message implosion at the source as long as the chance of packet loss is lower than that of successful packet delivery. In a NAK protocol, when a packet loss occurs close to the source, most of the receivers will detect the loss and send out NAK messages. The mere amount of these NAKs can easily result in implosion at the source.

One common approach to avoid feedback implosion at the source site is feedback suppression. In this scheme, a VAS agent, called NAK suppression agent, is set up at internal nodes in a multicast tree. NAK messages are unicasted from the receiver to its nearest ancestor node hosting a NAK suppression agent in the multicast tree. The ancestor node then forwards one NAK message to its nearest agent-hosting ancestor node and suppresses all duplicate NAKs.

NAK suppression agents are helpful for implosion prevention which is a key issue in reliable multicast. At the same time, however, they incur memory and processing overhead on routers. A NAK suppression agent must store sequence number information for each outstanding NAK message to suppress future duplicate NAKs. NAK messages are extracted from the IP fast forwarding path for a more detailed processing at the router where the NAK suppression agent is activated. The NAK suppression agent examines every received NAK to decide whether to forward it or to suppress it. Moreover, in order to eliminate the security vulnerability of false NAKs, the NAK suppression agent needs to deploy some authentication mechanism [8].

1.2 DDoS Defense

DoS attacks work by flooding some resource (a remote server or network) with large amounts of traffic, thereby preventing legitimate users from accessing that resource. A DDoS attack is a type of DoS attack where the attack traffic originates from multiple sources. (D)DoS attacks are threatening the utility of the Internet severely [9]. There has been a substantial amount of research work on defending against (D)DoS attacks.

The goal of IP traceback [5,6] is to construct the *attack tree* of a DDoS attack, which is composed of the network paths from attack sources to the victim. In practice, wily attackers can counterfeit extra routers into the traceback path [5] and IP traceback may be only partially deployed in the network [10]. Because of these practical limitations, the current IP traceback techniques can only construct an approximate or incomplete attack tree for a DDoS attack. Inaccurate attack trees are still valuable to DDoS defense as the defense measures such as packet filtering can be applied closer to the attack sources. Due to the possibility of source IP spoofing [11], the effective way to identify attack traffic is based on destination IP addresses. Blocking attack traffic based on destination IP addresses usually incurs *collateral damage*, that is, blocks the innocent traffic destined to the victim. Therefore, a better DoS defense measure is rate-limiting, instead of blocking, the attack traffic.

Pushback [7] is a cooperative mechanism in which a router can ask upstream routers to rate-limit DoS attack traffic. Given an attack tree constructed in IP traceback process, pushback mechanism can be used to (1) determine the rate limits for the attack traffic at different routers in the attack tree and (2) decide when to stop the rate-limiting process. In pushback, a VAS agent, called aggregate-based congestion control (ACC) agent, is activated at the routers in an attack tree. The ACC agent at a router periodically reports local status to the nearest ancestor ACC agent in the attack tree through a *pushback feedback message*. After combining the feedback from the nearest descendent ACC agents and local status, the ACC agent calculates/updates the rate limits for the attack traffic at the current router and the descendent routers, and then informs the ACC agents at those descendent routers.

Similar to NAK suppression agents, ACC agents incur memory and processing overhead on routers. ACC agents keep track of the status of attack traffic and reconsider rate limiting decisions periodically to update the rate limit for attack traffic. For each arriving packet, ACC agents need to check whether that packet belongs to attack traffic, and if so, forward or discard the packet according to the rate limit for the attack traffic.

1.3 Load-Balanced Agent Activation

In either reliable multicast or DDoS defense, a key problem is to decide where to activate VAS agents (NAK suppression agents or ACC agents) in a tree topology (multicast tree or attack tree). On one hand, a trivial approach which activates a VAS agent at every router in the tree meets VAS performance requirements (prevent implosion at the reliable multicast source or protect the victim under DDoS attack from malicious traffic), but leads to excessively high total memory and processing overhead on the routers. On the other hand, activating VAS agents at just a few routers in the tree reduces the total overhead, but may fail to satisfy the performance requirements, or even worse, may overload some routers with excessive feedback messages, thereby degrading the performance of packet forwarding for all traffic through those routers.

In the context of multiple trees, the situation becomes even more complicated. If many multicast/attack trees pass through a router, the trivial approach mentioned above will activate a VAS agent for every tree and the resultant memory requirement could overload the router. A naive solution to avoid the memory overload on routers is to deactivate the VAS agent for a randomly chosen tree at the overloaded router. However, such an approach has a deficiency that the casual selection of the tree being "dropped" may impair the VAS performance or overload other routers in that tree with excessive feedback messages.

Reaching a compromise between the VAS performance and the overhead of VAS agents is a complex problem. In this paper we explore a simplified version of that problem, referred to as Load-Balanced Agent Activation Problem (LBAAP). Specifically, the LBAAP problem is

> how to determine the number and placement of VAS agents in order to satisfy VAS performance requirements, with minimal total memory and processing overhead on routers and without overloading any router.

We examine the LBAAP problem in different contexts, propose corresponding algorithms, and evaluate the performances of various approaches by simulation.

1.4 Paper Organization

The rest of this paper is organized as follows. In Section 2, we define and analyze the LBAAP problem. In Section 3, we study the LBAAP problem and propose algorithms for both single tree case and multiple tree case. In Section 4, we evaluate the performances of various VAS agent activation approaches by simulation. In Section 5, we discuss the limitations of our algorithms and possible extensions. We survey related work in Section 6. Finally, we conclude the paper in Section 7.

2 Preliminaries

In this section, we introduce the models, definitions, and assumptions used in this paper.

In the context of either reliable multicast or DDoS defense, an *agent tree* structure can be constructed for describing the relationship among the involved entities. In the agent tree of a reliable multicast session, the leaves represent multicast group receivers and the internal nodes represent routers with NAK suppression capability. The root of the agent tree corresponds to the edge router at the multicast session source site. In the context of DDoS defense, the nodes in the agent tree represent the routers which support ACC agents. The root of the tree corresponds to the edge router connected to the DDoS

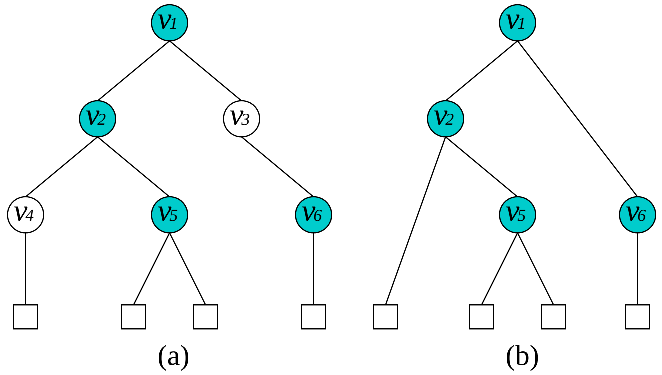

Fig. 1. (a) Multicast tree. (b) Agent tree.

victim. If we assume that all the routers in the network support value-added services, then the agent tree overlaps the underlying multicast/attack tree. Otherwise, it overlays the multicast/attack tree. As an example, Figure 1 illustrates the relationship between a multicast tree and the corresponding agent tree. The internal nodes in deep color represent routers with NAK suppression capability. Those NAK suppression capable routers and receivers constitute an agent tree which overlays the multicast tree. Since the discussion in this paper is based on the agent tree structure defined above, we will simply refer to an agent tree in reliable multicast or in DDoS defense as a multicast tree or an attack tree, respectively.

Given an agent tree, we assume a VAS agent is always activated at the root and each leaf sends out messages up the tree. We assume that the multicast source expects to receive one single NAK for a single packet loss event. Hence a NAK suppression agent is always activated at the root of the multicast tree in order to guarantee that. As to DDoS defense, we assume that ACC agents are always activated at all the leaves and the root of the attack tree. The ACC agent at the root controls the whole DDoS defense scheme on behalf of the victim. Activating ACC agents at the leaves is to maximize the efficacy of DDoS defense since the attack traffic is throttled furthest from the victim. Therefore, the internal nodes except the root of an agent tree has the option to activate or deactivate VAS agents. Each leaf and activated agent in the agent tree sends a feedback message to its nearest ancestor node hosting an agent.

We also assume that a VAS agent supports only one application instance (agent tree) and multiple agents need to be activated to handle multiple application instances. The majority of the memory overhead introduced by a VAS agent is the amount of memory recording the application state information; the memory foot print of the agent program itself is fixed and negligible. Hence, having an agent support a single or multiple application instances makes little difference to the problem discussed in this paper.

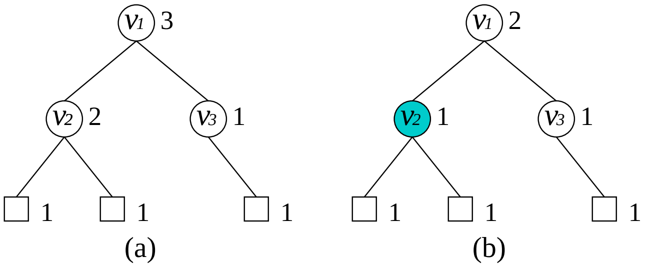

Fig. 2. (a) No agent is activated at any internal node. (b) An agent is activated at internal node v_2.

The number of the children of a particular node in a tree is called the *degree* of that node. If a VAS agent is activated at a node in an agent tree, the degree of that node reflects the lower bound of the number of feedback messages received by that node. In reliable multicast, the *worst case* occurs when a single packet loss near the multicast session source makes every receiver emit a NAK message toward the source. If a NAK suppression agent is activated at an internal node in a multicast tree, then the number of NAK messages received by that node in the worst case will never be less than the degree of the node. In DDoS defense, each ACC agent sends out a pushback feedback message up the attack tree every *pushback refresh period*. If an ACC agent is activated at a node in an attack tree, the degree of the node gives the minimal number of pushback feedback messages received by that node during a pushback refresh period. Consider an internal node v hosting an agent in an agent tree: the more the agents activated at its children, the less the feedback messages received by node v. If an agent is activated at each of the children of v, then the number of feedback messages arriving at node v is the same as its degree. For example, in Figure 1-b, $degree(v_1) = 2$; if activating agents at v_2 and v_6, the number of feedback messages received by v_1 will be also 2; otherwise, the number will be larger than 2.

The *weight* of a tree node is defined as follows: if the node is a leaf, its weight is 1; if the node is an internal node hosting an agent, its weight is 1; otherwise its weight is the sum of the weights of all of its children. We can regard the weight of a node as the number of feedback messages sent up the tree from/via that node. Suppose an internal node v is not hosting an agent and its weight is w. If a VAS agent is activated at node v, then the number of feedback messages received by node v will be w. In reliable multicast, node v will receive w NAKs in the worst case. In DDoS defense, node v will receive w pushback feedback messages every pushback refresh period. The definition of weight is illustrated in Figure 2. The number beside each node in the agent tree is the weight of the node. In Figure 2-a, no agent is activated in the tree, so $weight(v_2) = 1+1 = 2$. In Figure 2-b, an agent is activated at node v_2, so $weight(v_2) = 1$, and the number of feedback messages received by v_2 is 2.

We define the *processing overhead of an agent* to be the number of feedback messages received by the agent. For example, the processing overhead of an NAK suppression agent is the number of NAKs received in the worst case; and the processing overhead of an ACC agent is the number of pushback feedback messages received every pushback refresh period. The *processing overhead on a router* is the sum of the processing overhead of all agents activated at the router. We assume that the *memory overhead introduced by an agent* is constant with a value of 1. Hence the *memory overhead on a router* is the number of agents activated at the router. Given an agent tree, each leaf and activated agent in the tree sends one feedback message to its nearest ancestor node hosting an agent. Therefore the total processing overhead equals the sum of the number of leaves and the number of agents. Thus, minimizing the number of agents is equivalent to minimizing the total memory and processing overhead on routers.

The *memory load bound* on routers specifies the amount of the memory devoted for the VAS functionality, i.e., how many VAS agents can be activated at a router. The *processing load bound* indicates the number of arriving feedback messages which a router can afford. The processing load bound reflects not only the amount of computing resources available for value-added services at a router, but also the amount of link bandwidth available for receiving feedback messages.

Given certain memory and processing load bounds on the routers through which an agent tree passes, locating routers to activate VAS agents without exceeding the specified bounds for memory and processing capabilities prevents overloading routers and achieves load balancing among routers.

In this work, we assume that agent tree topologies are already known. Typically, Internet service providers (ISPs) have access to multicast/attack tree information within their domains. Our work concerns VAS agent activation within a domain. If a multicast/attack tree spans several domains, our algorithms can be used to find an activation of agents in each domain, for the portion of the tree which spans that domain. In addition, several approaches, such as *tracetree* [12] and *FIT* [13], have been proposed to efficiently and effectively collect multicast/attack tree topologies in the Internet.

3 Load-Balanced Agent Activation Problem (LBAAP)

We examine the LBAAP problem in both single tree and multiple tree cases and propose algorithms to solve the problem in these cases.

3.1 Single Tree Case

In the context of a single tree at most one VAS agent needs to be activated at a router. As long as the memory load bound is not set to be 0, the memory overhead introduced by an agent will not overload the router. Thus, we do not consider the memory overhead issue in the single tree case. Since the degree of an internal node is the lower limit of the processing overhead introduced by an agent on the node, we assume the processing load bound is never set to be smaller than the degree of any internal node in an agent tree.

The LBAAP problem in single tree case can be defined as follows. The input consists of an agent tree T rooted at r and a processing load bound PB. The processing overhead on an internal node v is represented as $pd(v)$. The goal is to select a set of internal nodes, R, to activate a VAS agent at each node in R, satisfying the following conditions:

(1) $r \in R$,
(2) $\forall v \in R, pd(v) \leq PB$, and
(3) the size of R is minimal.

We first show a canonical activation of agents, then present an algorithm to find such a solution in linear time. Given an optimal activation of VAS agents in a tree with a processing load bound PB, we can transform this activation into another optimal activation which satisfies the following two conditions:

(1) For any internal node v which hosts an agent but whose parent p does not host one, if we deactivate the agent at node v and activate an agent at v's parent p, then the processing overhead introduced by the agent on p will exceed the processing load bound PB.
(2) For any internal node v which hosts an agent and has siblings, the processing overhead on node v is not smaller than the weight of any of its siblings.

Given an optimal agent activation in a tree with processing load bound PB, we can transform this activation as follows. First we move each agent as high in the tree as possible without violating the load constraint. In other words, if an agent is activated at a node v but no agent is activated at v's parent p, then the following statement must be true. If we do not activate the agent at v but activate an agent at its parent p instead, the processing overhead on p, which is the same as the weight of p when no agents are activated at p and v, will exceed PB. Otherwise, we can do such transformation to obtain a new solution with the same number of agents. Second, we move each agent from the node where it is located to a sibling node whose weight is larger than the processing overhead introduced by the agent on the current node, if possible. In other words, if we activate an agent at a child u of a node p, but

```
/* T is an agent tree rooted at $r$ */
/* PB is a processing load bound */
LBAAP (T, PB) {
   pd($r$) := ActivateAgent($r$)
   activate a VAS agent at $r$
}

/* activate VAS agents in the subtree rooted at $v$ */
/* return $weight(v)$ after the activation */
ActivateAgent($v$) {
   if $v$ is a leaf node then {
       return weight($v$)
   } else {
       weight($v$) := 0
       for each child $u$ of $v$ do
           weight($v$) := weight($v$) + ActivateAgent($u$)
       while weight($v$) > PB do {
           select an internal node child $u$ of $v$ with largest weight
           activate a VAS agent at $u$
           pd($u$) := weight($u$)
           weight($v$) := weight($v$) - weight($u$) + 1
           weight($u$) := 1
       }
       return weight($v$)
   }
}
```

Fig. 3. LBAAP algorithm for the single tree case.

not at another child v of p, then it must be the case that $pd(u) \geq weight(v)$. Otherwise, we do such transformation and the weights of all ancestors of node u remain the same or decrease, and then we obtain a new solution with the same number of agents.

Our algorithm works as below. Given an agent tree T and a processing load bound PB, we process tree T in a bottom-up approach. For every internal node v, if $weight(v) > PB$, select a child node, u, which is an internal node with the largest weight among all the children of v, and activate a VAS agent at u. After that activation, $weight(v)$ reduces to $weight(v) - weight(u) + 1$, and $weight(u)$ becomes 1. We repeat the operation above until $weight(v) \leq PB$, then go to the next node. The correctness of the algorithm follows from the argument above on the canonical activation of agents. The running time of the algorithm is $O(n \cdot d)$, where n is the number of the internal nodes in the tree, and d is the average degree of all internal nodes. Figure 3 shows the pseudocode of the algorithm.

3.2 Multiple Tree Case

In this section we consider the LBAAP problem in the context of multiple trees. In the multiple tree case, multiple multicast/attack trees exist in the network and some routers appear in several trees simultaneously. Consider multiple agent trees pass through a router v. If the memory load bound is smaller than the number of trees, router v cannot activate a VAS agent for every tree, so it has to choose some of the trees for which it would not offer value-added service. A random selection of the tree being "dropped" from router v may make excessive feedback messages flow to some ancestor router in that tree. That may overload the ancestor router with excessive feedback messages. Thus, an intelligent selection of the trees being served is an important issue. Since a VAS agent is always activated at the root of an agent tree, we assume that no router is the root of more than MB agent trees, where MB is the memory load bound. In other words, a router is connected to at most MB reliable multicast sources or at most MB victims under DDoS attack.

The LBAAP problem in multiple tree case can be defined as follows. The input consists of a graph $G = (V, E)$, with V denoting the set of nodes and E denoting the set of edges connecting the nodes, and a set $T = \{T_1, T_2, \ldots, T_m\}$ of m agent trees in G. Let $V_r = \{v \in V$ and v is an internal node of $T_i(1 \leq i \leq m)\}$. Each tree $T_i \in T$ is rooted at $r_i \in V_r$. The memory load bound is MB, and the processing load bound is PB. The memory overhead on a node v is represented as $md(v)$, and the processing overhead on v is represented as $pd(v)$. The goal is to select a set of nodes $R \subseteq V_r$, and for each node $v \in R$, activate n_v ($n_v \geq 1$) agents on it, satisfying the following conditions:

(1) $r_i \in R$, for $1 \leq i \leq m$,
(2) $\forall v \in R, md(v) \leq MB$,
(3) $\forall v \in R, pd(v) \leq PB$, and
(4) $\sum_{v \in R} n_v$ is minimized.

As we will see in Section 3.3, even in a simplified model, the LBAAP problem in multiple tree case turns out to be much harder than the single tree case. We conjecture it is NP-hard based on the fact that a minor variation of the multi-tree LBAAP problem is NP-hard (discussed in the next subsection). Since we believe the multi-tree LBAAP problem is NP-hard, rather than computing an optimal solution, in this paper we focus on computing a feasible solution. Specifically, we propose a heuristic to solve the LBAAP problem in multiple tree case.

First at all, we define an operation on trees, called *node removal.* Removing a node v from a tree T means to remove node v from tree T and make all the children of v to be the children of v's parent. The root of a tree can not be

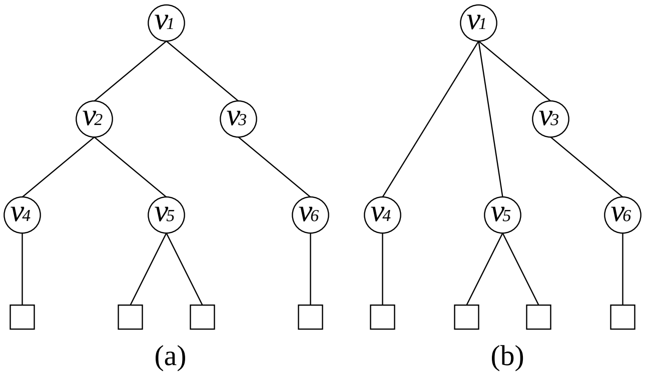

Fig. 4. (a) Before removing node v_2. (b) After removing node v_2.

removed. An example is shown in Figure 4. In practice, removing a node v from a tree T corresponds to that router v would not offer value-added service to tree T.

The heuristic proceeds in two steps:

(1) *Tree Selection:* For each node v which appears in more than MB trees as an internal node, we select MB trees to be served by v. The selection is based on the potential processing overhead on the parent of v in each tree if a VAS agent is not activated at v for that tree. Let N_v denote the number of the trees in which a node v appears as an internal node. The selection algorithm is as below. For each node $v \in V_r$, if $N_v \leq MB$, then all N_v trees are selected as candidates which are supported by node v with VAS agents. Otherwise we need to select MB out of N_v trees in the following way.
   (a) For each tree T_i passing through v, if v is the root of T_i, select T_i as a candidate.
   (b) After step (a), if the number of candidates is smaller than MB, then for each of the remaining trees, remove node v from the tree. Arrange these trees into a list in decreasing order according to the degree of the parent of node v in each tree. Select trees from the beginning of the list as candidates until the number of candidates increases to MB. For the trees selected as candidates, undo the operation of removing node v and restore them to their previous topologies.

(2) *Processing Load Bound Assignment:* After making the tree selection for each node $v \in V_r$, each tree remains the same size or it changes to a smaller size tree, by removing some internal nodes. We assign the processing load bound $L = \frac{PB}{MB}$ to each tree. Then, we apply the LBAAP algorithm for the single tree case individually on each tree.

Step 1 makes sure that each router supports no more than MB agent trees, and then the memory load bound is satisfied. Step 2 makes sure the processing overhead on each router is not larger than PB, and then the processing load bound is satisfied.

When the processing load bound L is smaller than the degree of some internal node in a tree, the LBAAP algorithm for single tree case can not produce a solution for agent activation in that tree. When the processing load bound on routers, PB, is set to be smaller, the processing load bound L assigned to each tree becomes smaller. When the memory load bound on routers, MB, is set to be smaller, more nodes are removed from each tree, and the degrees of some nodes in the trees become larger. Thus, when the memory and processing load bounds on routers are set to be too small, the LBAAP heuristic for the multiple tree case may not produce a solution even if there exists one.

3.3 Hardness of Multi-tree LBAAP Problem

We conjecture the LBAAP problem in multiple tree case is NP-hard, as a minor variation of the multi-tree LBAAP problem is NP-hard.

Consider a graph $G = (V, E)$, with V denoting the set of nodes and E denoting the set of edges. Let $T = \{T_1, T_2, \ldots, T_m\}$ be a set of m agent trees in G. Let $V_r = \{v \in V$ and v is an internal node of $T_i (1 \leq i \leq m)\}$. Each edge in E appears in some tree T_i and no edges are shared by any two different trees, but multiple trees may share the same node in V. Multiple VAS agents are allowed to be activated at a node $v \in V_r$ and one agent can support multiple trees, but each tree passing through node v can be associated with at most one agent at v. There is no memory load bound or processing load bound on nodes. Instead, we set the processing load bound L on agents. The objective is to activate as few agents as possible such that no agent has a processing overhead more than L. We call this problem V-LBAAP. Clearly, this is a variant of the multi-tree LBAAP problem.

Since the trees in T do not share edges, we can treat each tree $T_i \in T$ independently and compute an optimal agent activation for T_i using the LBAAP algorithm for the single tree case. Suppose we already finished executing the LBAAP algorithm for each tree in T. Then, at each node v, the number of agents is k_v $(0 \leq k_v \leq m)$ such that each agent supports just one tree. In the V-LBAAP problem, an agent may support multiple trees as long as its processing overhead does not exceed L. So we would like to minimize the number of agents at each node by making single agent support multiple trees without violating the load constraint, thereby minimizing the total number of agents.

The task of minimizing the number of agents at a node can be formulated as follows. The input consists of a finite collection of n integers $d_1, \ldots, d_n$, and an upper bound L $(L \geq d_i$, for $1 \leq i \leq n)$. These integers represent the processing overheads of the agents each of which supports just one tree. L is the processing load bound on agents. The goal is to decide the minimum number of groups into which the integers can be assigned such that the sum of the integers in each group does not exceed the bound L. This is precisely the integer bin packing problem which is known to be NP-hard [14].

4 Evaluations

We evaluate the performances of various VAS agent activation approaches by simulation in the context of both a single tree and multiple trees. To the best of our knowledge, the LBAAP problem has not been addressed before. So we compare the algorithms presented in the previous section with some approaches proposed by ourselves.

The metrics used to evaluate the performance are total memory overhead and total processing overhead on routers. In the single tree case, as we mentioned in Section 2, the total memory overhead equals the number of VAS agents, and the total processing overhead equals the sum of the number of leaves and the number of agents. In the multiple tree case, the total memory overhead is the sum of the total memory overhead in all trees, and the total processing overhead is the sum of the total processing overhead in all trees. In other words, the fewer the VAS agents, the lower the total memory and processing overhead.

Because the research on DDoS defense is at an early stage, there is no data available on attack tree topologies. So we experiment on activating NAK suppression agents in multicast trees in simulation. We believe we would get the similar simulation results in the context of DDoS defense.

4.1 Single Tree Case

For the single tree case, we evaluate three VAS agent activation approaches. The first approach is the LBAAP algorithm for the single tree case, presented in Section 3.1. The second approach, called *full deployment* (FD) approach, is to activate a VAS agent at every internal node in the agent tree. And the third approach, called *branching point deployment* (BPD) approach, is to activate a VAS agent at every branching point in the agent tree.

The simulation is based on the real multicast tree topologies collected by Chalmers *et al.* using *mwalk* [15]. Table 1 shows the multicast trees used in the simulation. Each tree is presented with the number of internal nodes, the

Tree	# of Internal Nodes	# of Leaves	Highest Degree
1	163	50	6
2	207	49	4
3	255	99	7
4	271	100	5
5	427	247	11
6	453	248	11
7	587	497	32
8	598	496	30
9	762	993	71
10	774	992	71

Table 1
Multicast Trees.

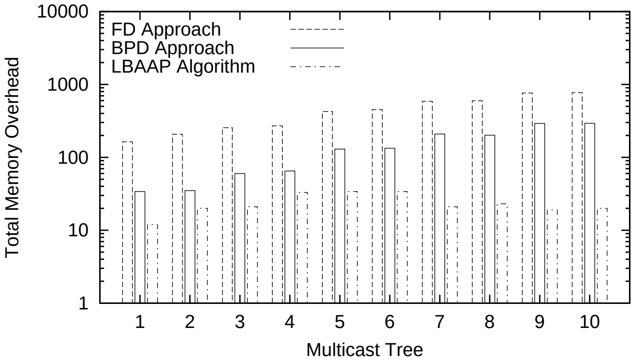

Fig. 5. Single Tree Case: Total Memory Overhead.

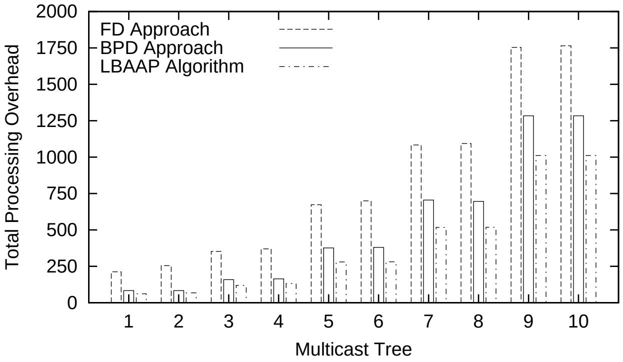

Fig. 6. Single Tree Case: Total Processing Overhead.

number of leaves, and the highest degree of all nodes. We set the processing load bound on routers to be the highest degree of all nodes, because none of these three approaches can find a feasible solution for agent activation with a lower processing load bound.

Figure 5 shows the results of the total memory overhead incurred by these three agent activation approaches. X-axis denotes the multicast tree used in the simulation, and Y-axis denotes the total memory overhead on routers. The Y-axis is in logarithmic scale. The FD approach results in the highest total memory overhead, the BPD approach results in lower overhead, and the LBAAP algorithm results in the lowest overhead with a significant difference compared to the other two approaches. Figure 6 shows the results of the total processing overhead. We have a similar observation: the LBAAP algorithm results in the lowest total processing overhead among these three approaches. These results demonstrate that the LBAAP algorithm for the single tree case leads to smaller total memory and processing overhead on routers compared with the FD and BPD approaches.

4.2 Multiple Tree Case

For the multiple tree case, we compare the LBAAP heuristic, presented in Section 3.2, with two alternative agent activation approaches. These two approaches are the extended FD and BPD approaches for the single tree case, respectively. The *extended FD* (EFD) approach is to activate a VAS agent at every internal node in each agent tree, and deactivate randomly chosen agents at routers when the memory overhead on the routers exceeds the memory load bound. The *extended BPD* (EBPD) approach is to activate a VAS agent at every branching point in each agent tree, and deactivate randomly chosen agents at memory overloaded routers.

The solution given by the LBAAP heuristic is always a feasible solution for agent activation, though the LBAAP heuristic can not produce a solution when the memory and processing load bounds are too small. The other two approaches always produce a solution no matter what values the memory and processing load bounds are, but the solution may not be feasible. These two approaches only consider memory load bound, omitting the processing load bound. A random selection of the agent being deactivated may make excessive feedback messages flow to some ancestor router in the agent tree so that the processing overhead introduced by the agents on that router exceeds the processing load bound.

In the multiple tree case, the simulation needs to be based on multiple trees within a network. The multicast trees generated by *mwalk* are rooted at the same router. Contrary to the single tree case, the *mwalk* data can not represent a realistic scenario for the evaluation in the multiple tree case. Because of this reason, we use synthetic multicast tree topologies in the simulation. We create ten transit-stub graphs using the Georgia Tech Internetwork Topology Models (GT-ITM) [16] to simulate network topologies. The sizes of those graphs range from 95 to 980 nodes. Given a transit-stub graph, we randomly choose one

Group	Net	SumR	SumL	N1	N2	N3	N4
1	95	152	101	5	13	19	16
2	180	306	207	11	13	39	38
3	280	404	228	10	37	42	48
4	380	547	353	23	67	62	51
5	475	724	490	12	63	95	75
6	580	717	527	43	96	110	38
7	680	1093	722	20	97	130	122
8	780	1169	805	43	95	132	135
9	870	1419	1003	18	95	162	181
10	980	1563	1047	37	113	188	184

Table 2
Multicast Tree Groups.

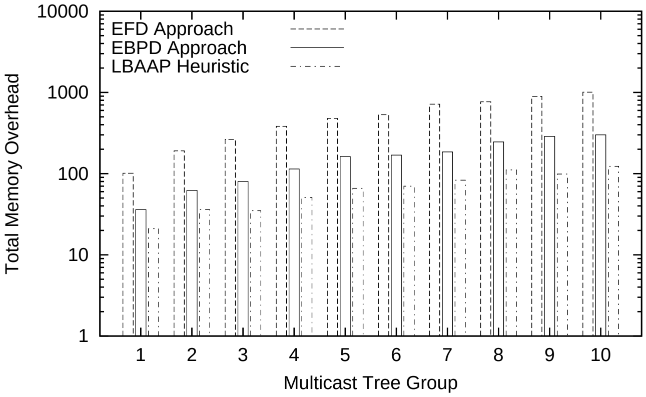

Fig. 7. Multiple Tree Case: Total Memory Overhead (4 Multicast Trees, Memory Load Bound = 2).

node as multicast source and multiple nodes as receivers of a multicast group from stub domains in that graph. We use the Network Simulator (ns-2) [17] to simulate the multicast session and extract the multicast forwarding path as a multicast tree over the network topology. For each network topology, we generate four multicast trees to form a group as the input for the agent activation procedure. Table 2 shows the multicast tree groups used in the simulation. The first column in the table is the group number, the second column is the size of the network from which a group of trees are created, the third column is the sum of the internal nodes of all trees in a group, the fourth column is the sum of the leaves of all trees in a group, and the remaining columns are the numbers of routers in the network through which 1, 2, 3, and 4 multicast trees pass, respectively.

In the simulation, we set the memory load bound on routers to be 2 and 3, respectively. For each memory load bound, we set the processing load bound to be the smallest value such that the LBAAP heuristic can produce a solution. Figures 7 and 8 show the results of the total memory overhead incurred by

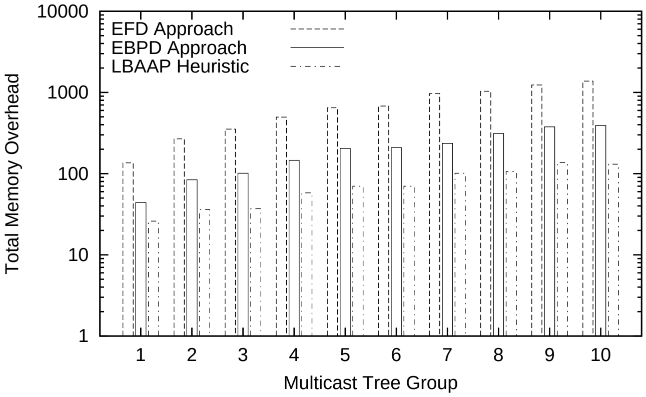

Fig. 8. Multiple Tree Case: Total Memory Overhead (4 Multicast Trees, Memory Load Bound = 3).

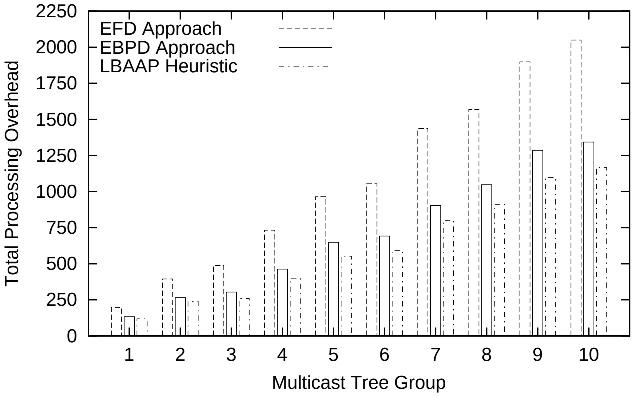

Fig. 9. Multiple Tree Case: Total Processing Overhead (4 Multicast Trees, Memory Load Bound = 2).

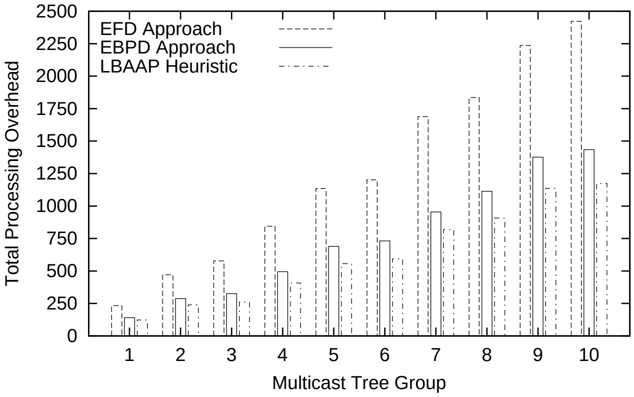

Fig. 10. Multiple Tree Case: Total Processing Overhead (4 Multicast Trees, Memory Load Bound = 3).

these three agent activation approaches, with the memory load bound set to be 2 and 3, respectively. X-axis denotes the multicast tree group used in the simulation, and Y-axis denotes the total memory overhead on routers. The Y-axis is in logarithmic scale. The EFD approach results in the highest total memory overhead, the EBPD approach results in lower overhead, and the LBAAP heuristic results in the lowest overhead. Figures 9 and 10 show the results of the total processing overhead, with the memory load bound set to be 2 and 3, respectively. We observe similar results: the LBAAP heuristic results in lower total processing overhead than the other two approaches. These results demonstrate that the LBAAP heuristic for the multiple tree case leads to smaller total memory and processing overhead on routers compared with the EFD and EBPD approaches.

Since the EFD and EBPD approaches omit the processing load bound and work with randomness, for the same multicast tree topologies and memory load bound, each time the produced solution for agent activation is different, and may or may not satisfy the processing load bound requirement. For a given memory load bound, we execute both EFD and EBPD approaches for 20 times, each time collecting the highest value of the processing overhead on routers. For each memory load bound, we also run the LBAAP heuristic and collect the highest processing overhead on routers, with the processing load bound set to be the smallest value such that the LBAAP heuristic can produce a solution. Given a solution for agent activation, if the processing overhead on some router exceeds the processing load bound, then that router will be overloaded by excessive feedback messages and the solution is not feasible.

Figures 11 and 12 show the results of the highest processing overhead on routers. X-axis denotes the multicast tree group used in the simulation, and Y-axis denotes the highest processing overhead on routers. In the figures, the horizontal bar represents the lowest processing load bound such that the LBAAP heuristic can produce a solution; the point represents the highest processing overhead on routers in the solution produced by the LBAAP heuristic; finally, two vertical bars represent the range of the highest processing overhead on routers produced by the EFD and EBPD approaches, respectively. In Figure 11, the memory load bound is set to be 2, and the processing load bound is set to be the smallest value such that the LBAAP heuristic can produce a solution, which is always feasible. The EFD and EBPD approaches hardly produce a solution for agent activation wherein the highest value of the processing overhead on routers is less than the processing load bound. In Figure 12, the memory load bound is set to be 3, and the processing load bound is still set to be the smallest value such that the LBAAP heuristic can find a feasible solution. In most cases the EFD and EBPD approaches find a feasible solution, but sometimes fail to find a feasible solution.

Based on the simulation results, we can find that the (E)FD and (E)BPD

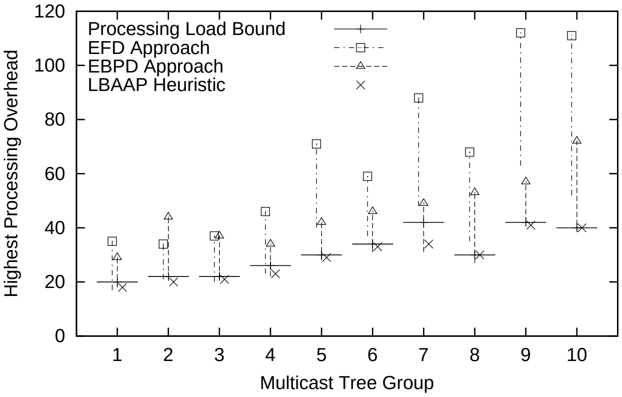

Fig. 11. Multiple Tree Case: Highest Processing Overhead (4 Multicast Trees, Memory Load Bound = 2).

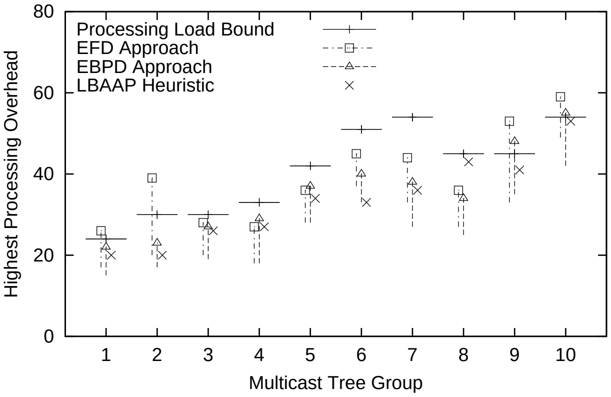

Fig. 12. Multiple Tree Case: Highest Processing Overhead (4 Multicast Trees, Memory Load Bound = 3).

approaches for VAS agent activation are simple and easy to implement, but lead to high total memory and processing overhead on routers. The VAS agent activation algorithms proposed in this paper are more complex than the other two approaches, but lead to lower total memory and processing overhead on routers. Furthermore, in the context of multiple trees, when routers do not have abundant memory and computing resources for VAS agents, that is, when the memory and processing load bounds are low, the EFD and EBPD approaches can not find a feasible solution for agent activation, whereas the LBAAP heuristic can still produce a feasible solution.

5 Discussion

The VAS agent activation algorithms proposed in this paper take agent tree topologies as the input. The deployment of these algorithms in practice can be done in a centralized manner. That is, the algorithms are implemented on a central server, which collects the agent tree topologies, invokes the algorithms, and then commands the relevant routers to activate VAS agents.

In this paper, we assume that multicast/attack tree topologies are known to us. Previous work [12,13] studied how to efficiently collect multicast/attack tree topologies in the Internet. The algorithms proposed in this paper can be executed in polynomial time in a centralized manner. It takes one more RTT for the central server to command the selected routers to activate agents. So, the time for agent activation process stays within acceptable limits.

In reliable multicast, the centralized manner may be inefficient if the multicast group is dynamic. When the multicast tree topology varies frequently, the overhead of the algorithms on computing resources and network bandwidth will increase noticeably. On one hand, in most applications that use reliable multicast, such as one-to-many reliable file transfer, the set of receivers is mostly static and is known to the source in advance. As a result, the tree topology of a reliable multicast session is relatively stable during the lifetime of the session. On the other hand, if the structure of a multicast tree changes partially, i.e., in a subtree, the agent activation algorithms can be applied to the subtree to compute a new solution for agent activation in that subtree. Through combining this new solution for the subtree and the old activation of agents in the rest of the tree, we can get a new solution for the whole tree. When applying the agent activation algorithms to a subtree instead of the whole tree, the overhead on both computation and bandwidth is reduced. For instance, a router v hosts an agent for a multicast session. When the agent receives more NAKs than its upper limit, v sends a request to the server to ask for recomputing the activation of agents in the subtree rooted at v. The server will collect the current topology of the subtree, invoke agent activation algorithms, and activate agents in the subtree.

In DDoS defense, the attack tree is relatively stable. Moreover, even an inaccurate attack tree is still useful for defending against DDoS attacks.

Several areas remain to be addressed in future work. One is to generalize the LBAAP problem by allowing different nodes to have different load bounds, and develop corresponding algorithms. Even in the context of a single tree, the generalized LBAAP problem is much harder. Another extension of our work is to develop a VAS agent activation approach which works in a distributed manner. That is, given a tree, each router makes decisions on agent activation locally, based on the information from its parent, children, and siblings in the tree.

6 Related Work

From a theoretical standpoint, the LBAAP problem resembles two well-known graph theoretic problems: the k-median problem and the facility location problem. Given a graph with n nodes, the k-median problem is to select k out of n nodes as service centers so as to minimize the sum of the cost of each node accessing its nearest service center. Tamir [18] studied the k-median problem in a tree topology and proposed an optimal algorithm. Li *et al.* [19] used a similar approach to optimally place web proxies in a tree topology with a web server at the root. Their objective is to minimize the overall latency in serving client requests from the leaves of the tree. Qiu *et al.* [20] studied the same problem in a graph topology and proposed various heuristics. Krishnan *et al.* [21] studied the problem of optimal placement of web caches. Their goal is to minimize the overall flow or the average delay by placing a given number of caches into network. Shah *et al.* [22] studied the k-median problem in the context of content-based multicast. They defined a filter placement problem as a variation of the k-median problem and provided two algorithms for optimal filter placement with the objective of minimizing mean total network bandwidth utilization and mean information delivery delay.

In the facility location problem, besides the cost of accessing the nearest service center (facility), there is also a cost of building a facility onto a node to make it become a service center. The objective is to find a solution (including both the number and locations of the facilities) of minimum total cost [23]. Guha *et al.* [24] introduced the Load Balanced Facility Location Problem wherein the constraint of having a minimum load on facility nodes is added to the original definition of the facility location problem. They proved that this version of the problem is NP-complete and presented a constant factor approximation algorithm for it.

With respect to practical aspects, Papadopoulos *et al.* [25] investigated the performance of reliable multicast under various deployment strategies of the supporting functionality. Many studies have been done on the placement of other kinds of functionality agents in the context of reliable multicast [26,27]. The objective of these works is to reduce the number of retransmissions, latency, and resource utilization. In contrast, our work is to activate a set of NAK suppression agents in the multicast tree topologies of currently existing reliable multicast sessions, minimizing the total overhead and satisfying the load constraints on routers. Moreover, our work is a first step in exploring the tradeoff between the performance and overhead of network-assisted DoS defense mechanisms.

7 Conclusion

In this paper, we have explored the relationship between the performance of value-added network services (VAS) and the overhead imposed on routers by the VAS agents realizing those services. In particular, we have discussed the load-balanced agent activation problem (LBAAP) derived from the context of both reliable multicast and DDoS defense. The goal of the LBAAP problem is to activate VAS agents in the network with a manner that not only satisfies performance requirements but also avoid load imbalance among routers. We have developed a polynomial running time algorithm for the LBAAP problem in single tree case, and have proposed a heuristic for the LBAAP problem in multiple tree case. Finally we have evaluated the performances of various approaches for VAS agent activation through simulation.

Reliable multicast and DDoS defense are among many Internet services that require or greatly benefit from the aid of various VAS agents located at routers. Those agents introduce non-trivial overhead on the routers hosting them. Reaching a balance between the performance of those services and the overhead incurred by the VAS agents is a task worth delving.

References

[1] S. Shenker and J. Wroclawski, General characterization parameters for integrated service network elements, Internet Engineering Task Force, RFC 2215, September 1997.

[2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, An architecture for differentiated services, Internet Engineering Task Force, RFC 2475, December 1998.

[3] B. Krishnamurthy, C. Wills, and Y. Zhang, On the use and performance of content distribution networks, in Proc. of SIGCOMM Workshop on Internet Measurement, San Francisco, USA, pp. 169–182, November 2001.

[4] K. Almeroth, The evolution of multicast: From the MBone to inter-domain multicast to Internet2 deployment, IEEE Network, vol. 14, no. 1, pp. 10–20, January/February 2000.

[5] S. Savage, D. Wetherall, A. Karlin, and T. Anderson, Network support for IP traceback, IEEE/ACM Transactions on Networking, vol. 9, no. 3, pp. 226–237, June 2001.

[6] A. Snoeren, C. Partridge, L. Sanchez, C. Jones, F. Tchakountio, B. Schwartz, S. Kent, and W. Strayer, Single-packet IP traceback, IEEE/ACM Transactions on Networking, vol. 10, no. 6, pp. 721–734, December 2002.

[7] R. Mahajan, S. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker, Controlling high bandwidth aggregates in the network, ACM SIGCOMM Computer Communications Review, vol. 32, no. 3, pp. 62–73, July 2002.

[8] J. Gemmell, T. Montgomery, T. Speakman, N. Bhaskar, and J. Crowcroft, The PGM reliable multicast protocol, IEEE Network, vol. 17, no. 1, pp. 16–22, January/February 2003.

[9] L. Garber, Denial-of-service attacks rip the Internet, IEEE Computer, vol. 33, no. 4, pp. 12–17, April 2000.

[10] C. Gong, T. Le, T. Korkmaz, and K. Sarac, Single packet IP traceback in AS-level partial deployment scenario, in Proc. of IEEE GLOBECOM, St. Louis, USA, November 2005.

[11] Computer Emergency Response Team, IP spoofing attacks and hijacked terminal connections, CERT Advisory CA-95. 01, January 1995.

[12] K. Sarac and K. Almeroth, Tracetree: A scalable mechanism to discover multicast tree topologies in the Internet, IEEE/ACM Transactions on Networking, vol. 12, no. 5, pp. 795–808, October 2004.

[13] A. Yaar, A. Perrig, and D. Song, FIT: Fast Internet traceback, in Proc. of IEEE INFOCOM, Miami, USA, pp. 1395–1406, March 2005.

[14] S. Skiena, The Algorithm Design Manual, Springer-Verlag, 1998.

[15] R. Chalmers and K. Almeroth, On the topology of multicast trees, IEEE/ACM Transactions on Networking, vol. 11, no. 1, pp. 153–165, February 2003.

[16] K. Calvert, M. Doar, and E. Zegura, Modeling Internet topology, IEEE Communications Magazine, vol. 35, no. 6, pp. 160–163, June 1997.

[17] K. Fall and K. Varadhan, ns Notes and Documentation, UC Berkeley, LBL, USC/ISI, and Xerox PARC, http://www.isi.edu/nsnam/ns.

[18] A. Tamir, An O(pn^2) algorithm for the p-median and related problems on tree graphs, Operations Research Letters, vol. 19, no. 2, pp. 59–64, March 1996.

[19] B. Li, M. Golin, G. Italiano, X. Deng, and K. Sohraby, On the optimal placement of Web proxies in the Internet, in Proc. of IEEE INFOCOM, New York, USA, pp. 1282–1290, April 1999.

[20] L. Qiu, V. Padmanabhan, and G. Voelker, On the placement of Web server replicas, in Proc. of IEEE INFOCOM, Anchorage, USA, pp. 1587–1596, April 2001.

[21] P. Krishnan, D. Raz, and Y. Shavitt, The cache location problem, IEEE/ACM Transactions on Networking, vol. 8, no. 5, pp. 568–582, October 2000.

[22] R. Shah, R. Jain, and F. Anjun, Efficient dissemination of personalized information using content-based multicast, in Proc. of IEEE INFOCOM, New York, USA, pp. 930–939, April 2002.

[23] K. Jain and V. Vazirani, Approximation algorithms for metric facility location and k-median problems using the primal-dual scheme and lagrangian relaxation, Journal of the ACM, vol. 48, no. 2, pp. 274–296, March 2001.

[24] S. Guha, A. Meyerson, and K. Munagala, Hierarchical placement and network design problems, in Proc. of IEEE Symposium on Foundations of Computer Science, Redondo Beach, USA, pp. 603–612, November 2000.

[25] C. Papadopoulos and E. Laliotis, Incremental deployment of a router-assisted reliable multicast scheme, in Proc. of International Workshop on Networked Group Communications, Palo Alto, USA, pp. 37–46, November 2000.

[26] S. Guha, A. Markopoulou, and F. Tobagi, Hierarchical reliable multicast: Performance analysis and placement of proxies, Computer Communications, vol. 26, no.18, pp. 2070–2081, December 2003.

[27] P. Ji, J. Kurose, and D. Towsley, Activating and deactivating repair servers in active multicast trees, in Proc. of Tyrrhenian International Workshop on Digital Communications, Toarmina, Italy, pp. 507–523, September 2001.