Application Level Aggregation and Multicast in Peer-to-Peer Systems

Djelloul BOUKHELEF † and Hiroyuki KITAGAWA †‡

† Computer Science Department, University of Tsukuba
‡ Center for Computational Sciences, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki. 305–8573

E-mail:boukhelef@kde.cs.tsukuba.ac.jp, kitagawa@cs.tsukuba.ac.jp

Abstract We propose in this paper the Application-Level Aggregation and Multicast (ALAM) service for fast information dissemination and aggregation over large-scale peer-to-peer systems. Basic tree-based approach builds a spanning-tree and uses it to disseminate and aggregate information to and from the network members. Besides single-point of failure, using the same tree for broadcast and aggregation results in high query latency and workload imbalance. In this work, we explore a novel dual-tree protocol inspired from the cardiovascular system. The key idea is to build pairs of interior-node-disjoint overlay trees ($\mathcal{A}$-tree and $\mathcal{V}$-tree) routed at the same node. Aggregate queries are disseminated via $\mathcal{A}$-tree in top-down manner. Answers are gathered along its dual $\mathcal{V}$-tree in bottom-up manner. To achieve low query latency and load balance, two dual trees are built such that nodes that are far from the root in $\mathcal{A}$-tree are placed close to the root in its dual $\mathcal{V}$-tree, and vice-versa. The dual-tree protocol is flexible and self-adaptive, and inherits its robustness from the self-healing capability of the underlying overlay. We evaluate our solution in the context of Chord overlays. Simulation results demonstrate the full scalability and efficiency of our design and confirm its advantage over existing tree-based protocols in terms of latency, maintenance cost, and workload balance.

Key words Structured P2P, Application-level Aggregation and Multicast, Dual-tree.

1 Introduction

Information dissemination and aggregation are essential building blocks in a wide range of distributed applications, such as: resources discovery and monitoring, performance tuning, network analysis and visualization. Additionally, aggregation is a key primitive in many, *if not all*, holistic operations that require scanning large portion of the stored data (*e.g.* queries over non-indexed data, keyword search and data mining, top-k queries).

Dissemination and aggregation are two essential phases in the whole aggregation process. Aggregation is the operation of gathering partial information from various providers in the system and building concise summaries (statistics) about some overall properties of the system and its components. Dissemination is the inverse operation that delivers messages carrying various instructions and parameters to all the nodes in the network.

Peer-to-peer (P2P) computing has emerged as a powerful and efficient paradigm for building scalable Internet-wide distributed applications. P2P applications manipulate large amount of data which is increasing at a fast rate . *So far*, researches in this field focus mainly on designing efficient and scalable protocols aiming to improve their query expressiveness and responsiveness. However, the area of holistic operations (*i.e.* large-scale aggregate query) is still not well explored.

Scalable and decentralized monitoring of large-scale P2P systems, network size estimation, and resources discovery are examples of services that rely heavily on the aggregation operation. The implementation of these services often requires the availability, at each node in the network, of aggregated information about some overall properties of the network and its components. Designing scalable management services that adhere entirely to the decentralized and dynamic nature of P2P systems is a very challenging. On one hand the decentralized requirement excludes the use of global or central knowledge authority. On the other hand, network dynamism, due to nodes that frequently join and leave the network at will (phenomenon known as *churn*), may compromise the robustness and validity of the aggregation process.

There exist two major approaches for distributed aggregation processing: *probabilistic* [1, 2] and *deterministic* [3, 4, 5, 6, 7, 8, 9, 10, 11, 12].

Probabilistic approach is proposed primarily for networks where the topology and resources locations are out of control (*e.g.* unstructured P2P). This approach makes use of gossip to exchange content over a randomly connected net-

work. Due to lack of deterministic query processing mechanisms in gossip-based methods, significant amount of duplicate messages is exchanged in order to improve the accuracy of answers and increase success rate. This makes gossip-based methods more robust to communication failures; however it may limits their scalability and efficiency in case of large-size networks.

Deterministic approach The idea of this approach is to shape the network into a loop free sub-graph (*i.e.* spanning tree) that connects a root node to each member of the network. Tree-shaped overlay is then used to broadcast and collect information to and from the overlay members. Tree-based approach is suitable for *duplicate-sensitive* aggregate functions, such as COUNT and SUM.

1.1 Problems with the tree-based approach

Tree-based approaches consider either building one single tree or multiple trees shared among the overlay members.

In fact, one single-tree is vulnerable to single-point of failure and risk of bottleneck that may arise at the root node, since all queries must transit by this node. Moreover, a single shared tree would not be optimally fair for all network members, while using multiple trees would be very costly [7].

In this research we are addressing two other problems that have not been dealt with previously, namely query latency and workload imbalance.

Latency issue First, the elapsed time between issuing an aggregate query and receiving the final answer might be very long. This is because query messages need to travel from the root down to leaf nodes and then return back to the root carrying answers. Typically, the query *round-trip* latency corresponds to two time the tree height. In practice, query latency might be very high, especially in large-size overlays where the tree happens to very deep.

Load balance issue Second, relatively a small fraction of interior-nodes undertake all the burden of communication and processing loads, *i.e.* propagate queries, and aggregate answers. The processing load is related to the branching factor of a node, that is, the number of direct descendants (children) it forwards the query to and aggregates answers from. As such, leaf nodes do not participate in any of these two tasks since they have no children. For example, in a binary tree, which may be very deep for large networks [6], approximately half of the nodes are leaves, hence they are without any additional workload.

In summary, using the same spanning tree to disseminate queries and collect answers has twice the latency and processing load compared to a simple broadcast or aggregation.

1.2 Proposed solution

We focus in this research on the problem of aggregation processing in structured P2P systems. Our aim is to provide methods for fast and robust aggregation query processing while fairly balancing the processing load among the members of the networked system.

Actually, aggregation and dissemination are complementary operations and are consequently implemented using same techniques. However, exiting methods focus, generally, on the efficiency of one operation at a time, *i.e.* either dissemination or aggregation; but not on the efficiency of both of them together.

We introduce in this research the Application-Level Aggregation and Multicast (ALAM) service intended for fast dissemination and aggregation of information over large-scale dynamic P2P overlay networks. The ALAM service is, mainly, targeting new and emerging applications that require *near* real-time processing of aggregate queries over data stored across the overlay network. Examples of such applications include resources discovery and monitoring service, collaborative network security, and Massive Multi-players Online Games. These applications require fast and efficient sensing of changes in the networked system (*e.g.* nodes state, available resources, game status and players locations) in a *near* real-time fashion.

To address the problems of query latency and workload imbalance in the tree-based, we are exploring in this research a novel dual-tree protocol inspired from the cardiovascular system. The key idea is to build pairs of *interior-node-disjoint* overlay trees (*i.e.* $\mathcal{A}$-tree and $\mathcal{V}$-tree) that are routed at the same node. Aggregate queries are disseminated along an $\mathcal{A}$-tree in top-down manner. Answers are gathered along its dual $\mathcal{V}$-tree in bottom-up manner. To achieve low query latency and good load balance, the two trees are built such that nodes that are far from the root in $\mathcal{A}$-tree are placed close to the root in its dual $\mathcal{V}$-tree, and vice-versa. Trees are built on-the-fly and their shapes adapt dynamically to changes in the network membership. Dual-tree design is flexible and easy to deploy. Overlay trees are built using underlying overlay's routing links. Tree maintenance relies solely on the self-healing property of the overlay and does not incur any additional maintenance overhead.

We sketch an implementation of the dual-tree protocol on top of the well-known Chord overlay [13]. Analytical and simulation results demonstrate the full scalability and efficiency of our dual-tree approach to support fast aggregate queries over large-scale P2P systems. Analysis shows that the maximum latency is bounded by the network diameter. For instance, the dual-tree protocol is able to achieve a reduction of approximately 50% in the query latency and achieve better load balance compared to other tree protocols [4, 8], especially for large network sizes.

2 Dual-tree protocol

In this section, we propose a novel *dual-tree* protocol to implement scalable ALAM services over structured P2P systems as viable solution to the latency and load imbalance issues in the tree approach.

The dual-tree design imitates the *circulatory system* in mammals, which plays the important role of distributing the blood over all the body through blood vessels. In its very basic design, the pathway of the blood in the body consists of a circuit of vessels (*arteries* and *veins*) centered at the heart. The heart pumps the oxygenated blood to body tissue through the arteries vessels. Veins are blood vessels that carry deoxygenated blood from the tissues back towards heart (from *Wikipedia*).

Starting from a given a (*root*) node, the idea of our dual-tree design is to build a pair of separate and unidirectional trees called *Artery*-tree and *Vein*-tree ($\mathcal{A}$-tree and $\mathcal{V}$-tree, for short). Similarly to the heart in the body, the root node is source and sink at the same time. Each pair of $\mathcal{A}$-tree and $\mathcal{V}$-tree is rooted at the same root node which also plays the role of heart in the body. The $\mathcal{A}$-tree is used to disseminate data and query, and the $\mathcal{V}$-tree to gather feedbacks and answers.

The quality metrics of our solution are related to performance (query latency), cost (load balance and maintenance overhead), and robustness against failures. To guarantee good performance level on the three metrics, we are exploring a novel *dual-tree* design that builds pairs of complementary trees. We say that an $\mathcal{A}$-tree and its dual $\mathcal{V}$-tree are complementary if they satisfy the two following properties. First, $\mathcal{A}$-tree and $\mathcal{V}$-tree are *interior-node disjoint*, that is, interior nodes in one tree are likely to be leaves in the other tree, and vice-versa. Second, the *root-to-node* and *node-to-root* paths on the two trees are asymmetric, that is, the path length from the root to any node on one tree is compensated by the path length from that node to the root on the other tree, such that their cumulative path length (*i.e. root-to-node-to-root*) is very small, and is quite the same for all the overlay nodes.

Achieving these two properties in a decentralized and dynamic systems is a very challenging problem. In our dual-tree model, we build $\mathcal{A}$-tree and $\mathcal{V}$-tree such that nodes that are far from the root in one tree are placed closer to root on the other dual tree. In the ideal case, our tree construction algorithms are able to produce trees that are interior-nodes-disjoint, except for tiny fraction of overlay nodes.

To ensure high degree of robustness, our-design makes use of the underlying overlay's unicast routing links to build *implicit* dynamic self-organizing instances of the dual-tree infrastructure for dissemination and aggregation. Additionally, our protocol relies on the self-healing property of the underlying overlay and does not incur any additional maintenance overhead. This feature makes our dual-tree design very fault-resilient and robust, and therefore very suitable for highly dynamic overlay systems which are affected by high rate of nodes churn.

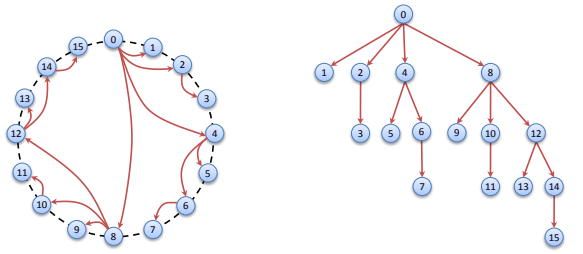
Finally, our new dual-tree design is lightweight and flexible. A dual-tree ALAM infrastructure can be easily deployed on top of existing overlays with reduced cost and time complexity and without altering their topologies.

In what follows we will describe how to build and maintain $\mathcal{A}$-trees and $\mathcal{V}$-trees in the context of ring-based overlay topology, namely Chord [13]. We will show how to process aggregate queries using our new dual-tree mechanism.

2.1 Construction of $\mathcal{A} - tree$

The $\mathcal{A}$-tree serves to propagate (disseminate) an information (*e.g.* aggregate queries, data) all over the networked system. An $\mathcal{A}$-tree is an overlay tree built implicitly by merging the shortest routing paths from the source node (root) to all the other nodes in the system. Shortest paths are generated using the "greedy" routing algorithm of the underlying overlay. The broadcast scheme in $\mathcal{A}$-tree is similar to the algorithm proposed in [4]. However, our method construct an $\mathcal{A}$-tree which is unidirectional, thus, does not maintain explicit parent-children relationship. As such, a node in the $\mathcal{A}$-tree is only *forwarder* of the query message to its children nodes in the overlay.

In general, a broadcast message contains the information to broadcast (*e.g.* query). The message is also tagged with a *limit* value that restricts the forwarding scope of each node. Upon receiving the query message, each forwarder node n propagates the message to all nodes in its finger table within the scope delimited by n itself and *limit*. Actually, the *limit* value helps eliminating duplicate messages.



(a) $\mathcal{A}$-tree rooted at 0 (b) Broadcast process using $\mathcal{A}$-tree

Fig. 1: Example of $\mathcal{A}$-tree rooted at node 0.

Starting from any root node n, the construction of $\mathcal{A}$-tree is the distributed process that recursively splits the key space into non-overlapping partitions using n's routing table. The first node in each partition, which is pointed by n's routing table, is delegated to farther propagate the query message to other nodes within its partition (forwarding scope). The distributed query dissemination process

is recursively repeated until nodes that receive the query message have no other nodes within their scopes to propagate the message to.

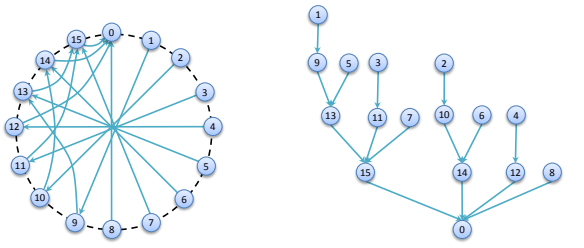
More specifically, when a node n receives a query message tagged with *limit* value, it forwards this message to each node in its routing table (*i.e.* $finger_i$) that is located between n and *limit* (in the clockwise direction).

Note that, initially, the scope of the root node is the whole space. Besides a copy of the query, the message sent to $finger_i$ is piggybacked with a new $limit_i$ of the partition which $finger_i$ will take care of. Generally, each $finger_i$ is responsible for the partition delimited by the $finger_i$ itself and the node pointed by the next entry in n's routing table, that is, $[finger_i, finger_{i+1})$.

Note that the set of children of a node n in the $\mathcal{A}$-tree contains all the nodes in its routing table that are within its forwarding scope (*i.e.* $[n, limit)$). Generally, the number of children, which represents also the branching factor of a node in the tree, is proportional to the size of the forwarding scope of a node.

2.2 Construction of $\mathcal{V} - tree$

The $\mathcal{V}$-tree serves to aggregate query answers from all over the network. A $\mathcal{V}$-tree is built in similar way to $\mathcal{A}$-tree but in bottom-up manner, by merging the shortest routing paths from each node in the overlay to the root node. Similarly to the $\mathcal{A}$-tree, $\mathcal{V}$-tree is a dynamic unidirectional overlay tree with implicit child-parent relationship. Every internal node in the $\mathcal{V}$-tree combine the received answers from its children with its own local answer and propagates it to the parent node until reaching the root. Thus, nodes in the $\mathcal{V}$-tree play the role of "*collector*" of answers.



(a) $\mathcal{V}$-tree rooted at 0 (b) Aggregation process using $\mathcal{V}$-tree

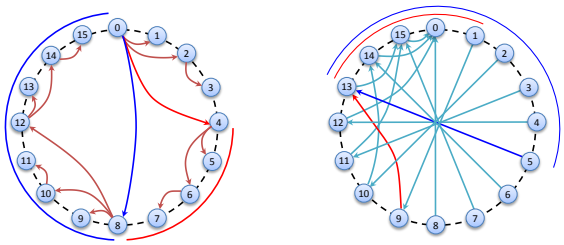
Fig. 2: Example of $\mathcal{V}$-tree rooted at node 0.

Note that in our dual-tree protocol, trees are not constructed explicitly. They are, however, implicitly built on-the-fly during query execution time. In opposition to the basic tree-based approach, our dual-tree approach does not materialize explicitly the parent-child relationship on the trees. More specifically, the parent (*resp.* child) of a node is implicitly determined by the next routing hop towards (*resp.* from) the root of the tree.

Let p and q two nodes in the overlay.

Property 2.1 (Parent-child relationship on $\mathcal{A}$-tree)**.** *We say that node q is a descendent of node p on $\mathcal{A}$-tree, if and only if p is on the routing path from the root to q. Node q is a child of node p if it is a direct descendent of p.*

Property 2.2 (Child-parent relationship on $\mathcal{V}$-tree)**.** *Similarly, a node p is an ascendent of node q on $\mathcal{V}$-tree, if and only if p is on the routing path from q to the root. Node p is the parent of q if it is a direct ascendent of q.*



(a) Forwarding scope (b) Aggregation scope

Fig. 3: Forwarding scope on $\mathcal{A}$-tree and $\mathcal{V}$-tree.

Figure 2 depicts the $\mathcal{V}$-tree rooted at 0. We can check that concatenation of tree vertices from any node to node 0 is exactly the routing path from that node to node 0. Note that node 8 is interior-node in $\mathcal{A}$-tree with 3 children, and a leaf node in $\mathcal{V}$-tree, and vice-versa for node 15.

2.3 Extension of Chord

Actually, each node in $\mathcal{V}$-tree needs to determine the set of *prospective children* from whom it will receive answers, hence, is should to wait from them. This situation is more complicated than in the $\mathcal{A}$-tree, as nodes determine their parents on $\mathcal{V}$-tree at runtime.

To solve this problem we propose to extend Chord protocol as follow. Each link from a node p to its i-th *finger* node $q = finger[i]$ is tagged with the *routing scope* of that link on the ring. Node q is called *contact* of the *anti-finger* node p on that portion of the ring (*i.e.* routing scope). For example, on figure 3a, 4 is contact of 0 for the interval $[4, 8)$, 9 is contact of 1 for $[9, 1)$.

Moreover, if the routing path of a query issued by a node p or is passing by it (*i.e.* p is intermediate) and targeting a node t in the routing scope for which a node q is contact on behalf of node p, this routing path should imperatively transit by node q. For example, on figure 3a if node 0 wants to route towards nodes 5 to 7, its must forward the query to node 4, because its the closest finger in node 0's routing table to these nodes. Actually, this extension of Chord protocol is very light and requires only one integer per link (*i.e.* entry in the routing table). This can be implemented, for instance, by piggybacking the information about routing scope to the periodic maintenance messages exchanges between neighboring nodes in the overlay.

Now, as for the construction of $\mathcal{V}$-tree, the routing scope of link helps determining if a node p is parent of node q on the $\mathcal{V}$-tree rooted at node r and, hence, it should wait for an answer from node q.

Intuitively: a node p is the parent of node q on the $\mathcal{V}$-tree rooted at node r, if and only if node q has a link towards node p, and node r is in the scope of that link. In addition, in a valid routing overlay, the parent of a node on the $\mathcal{V}$-tree exists and is unique.

2.4 Aggregation processing

Basically, when a node wants to compute an aggregate function it issues a message that encloses the aggregate query parameters. Processing the aggregation query consists first of disseminating the query to all the overlay nodes. Each node, then, processes the query over its local data and sends the answer directly or indirectly to the issuing node. In the later case, the local answer is combined with the answers from other nodes before it is shipped to the next node until reaching the issuing node. Finally, the issuing node compute the final answer using the partial answers it received from the other overlay nodes.

The aggregation process in our dual-tree follows also the same execution model. During an aggregation process, the root node broadcasts the aggregate query using its own implicit $\mathcal{A}$-tree. Forwarder nodes in the $\mathcal{A}$-tree progressively propagate the query message to all the other nodes in the overlay. Upon receiving the query message, each node evaluates the aggregate query over its local data. On the other hand, and as a collector on the $\mathcal{V}$-tree, each node determines the set of prospective children to wait answers from. When all the answers are received, the node compiles them with its local answer into one aggregate value, and forwards this answer to its parent on $\mathcal{V}$-tree. The final answer is compiled at the root node using partial answers received from its direct children on $\mathcal{V}$-tree.

3 Experiment

To validate our method, we evaluated the performance of the dual-tree design on a simulation prototype of Chord overlay. The quality metrics used in our simulation are related to performance (query latency and query stay-time per node) and cost (workload distribution).

We comparison purpose, we run the dual-tree procol against two tree-based techniques, namely: *direct single-tree* [4] and *reverse-single-tree* [9]. These two methods use similar tree construction mechanisms, and both of them were designed in the context of ring-based DHTs.

Settings We simulated Chord overlays with a maximum size of (2^m) nodes, for different values of m ($m = 4..16$). For a given value of m, we evaluate the performance for various *effective* network sizes (n), with n varying from 4 to 2^m. Nodes are uniformly distributed on the Chord ring. For each network size, thousands of aggregate queries are issued from randomly chosen nodes, and the average of the measured values is taken for each experiment.

3.1 Query latency

In this experiment we analyze the efficiency of our design in terms of total query latency, which measures the cumulative path length from *root-to-node* on a $\mathcal{A}$-tree and *node-to-root* on the $\mathcal{V}$-tree; which correspond to the cumulative number of hops it takes for the query message to reach a node and the answer to return back to the root.

Figure 4a depicts the maximum query latency as function of network size ($m = 12$). As we can see, the maximum query latency in our dual-tree protocol is smaller than in the two other methods. For relatively dense overlays (more than 75% of the nodes are present in the overlay), the query latency in the dual-tree approach converges rapidly to $\log n + 1$, while it is constantly increasing towards $2 \times \log n$ for the two other single-tree approaches.

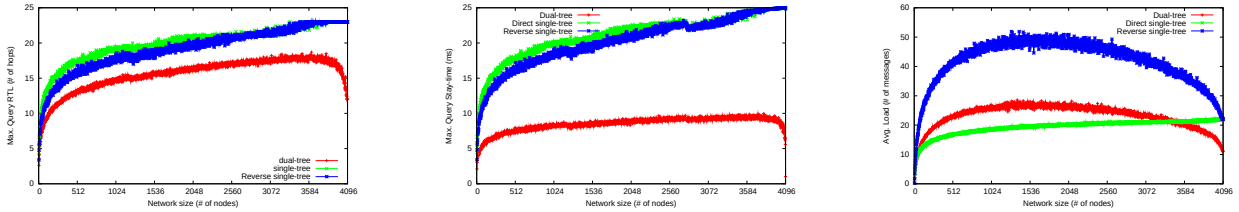
3.2 Query stay-time

The previous experiment gives insight about the total query execution time. To understand the flow of messages in the system, we break down the query execution time to measure how long does the query stay in each node. The *query stay time* refers to the period of time from receiving the query messages by a given node until it delivers its answer to its parent node. This time includes the query processing over local data as well as the waiting time for children's answers.

Actually, this measure gives an idea about the internal resources usage at each node as well as the degree of parallelism. In addition, the query stay time measure has a direct impact on the system performance and robustness. First, shorter stay time is expected to increase the system throughput, that is, the number of queries that can be executed by each node within the same period of time. Second, it provides better robustness against node churn if the query stay time is shorter than the node's lifetime.

In this experiment we simulated an overlay network considering different values for the link latency and local query processing time. Graphs on figure 4b depicts the maximum query stay time, with the network link latency set to 5 ms, and local query processing time to 1 ms.

Both the two measures, show that the query stay time in our dual-tree model is order of magnitude smaller compared to other tree-based methods. We can see also that the stay time is largely dominated by the query processing time. This is because, the query message reaches a node at approximately the same time with the children's answers.



(a) Query latency (b) Query stay time (c) Workload

Fig. 4: Performance of the dual-tree protocol (worst case).

3.3 Workload distribution

In this experiment we studied the distribution of processing workload among overlay nodes. The workload refers the cumulative number of query messages propagated and the answer messages gathered by each node.

Graphs on figure 4c depicts the maximum workload. In this experiment we consider only nodes with effective workload, that is, they broadcast the query or collect answers. In this experiment we omitted the local query processing load as it is a baseline for all nodes. We excluded nodes without effective load, *i.e.* nodes that are leaves in both $\mathcal{A}$-tree and $\mathcal{V}$-tree for our dual-tree protocol, and leaf node the single-tree methods.

We can see that the maximum workload is slightly larger compared to the direct single-tree method for relatively small overlays. It is however much smaller for highly dense network.

In summary, simulation results consolidate our theoretical findings and show that our dual-tree approach reduces notably the query latency and achieve fair workload distribution; which makes it suitable for fast and scalable aggregate query processing in large-scale P2P networks.

4 Conclusion and Future Work

In this research we introduced the Application-Level Aggregation and Multicast (ALAM) service, a comprehensive lightweight framework for scalable processing of holistic aggregation operations over data residing in a large-scale P2P system. To address the problems of latency and load imbalance in the tree-based approach, we presented and evaluated a novel dual-tree protocol inspired from the mammals' circulatory system.

The dual-tree protocol is fully decentralized and scalable. The proposed schemes allows for fast and robust information dissemination and aggregation over extremely large dynamic networks. Experimental and analytical results demonstrate the performance of the proposed dual-tree design and confirm its advantage in term of efficiency, workload balance. For instance, dual-tree design reduces the total query latency by approximately up to 50% compared to other tree-based methods [4, 9], especially for large-sized networks.

As future work, plan to study the robustness of the dual-tree protocol under churn. We plan to deploy the dual-tree protocol on other P2P schemes. We are also considering approximate online aggregate query processing.

Acknowledgment This study has been partially supported by Grant-in-Aid for Scientific Research on Priority Areas from MEXT (#21013004).

References

[1] P.T. Eugster, R. Guerraoui, S.B. Handurukande, P. Kouznetsov, and A.M. Kermarrec, "Lightweight probabilistic broadcast," ACM Trans. Comput. Syst., vol.21, no.4, pp.341–374, 2003.

[2] J. Leitão, J. Pereira, and L. Rodrigues, "Gossip-based broadcast," in Handbook of Peer-to-Peer Networking, pp.831–860, Springer, 2010.

[3] D. Bradler, J. Kangasharju, and M. Mühlhäuser, "Optimally efficient multicast in structured P2P networks," CCNC, pp.123–127, 2009.

[4] S. El-Ansary, L.O. Alima, P. Brand, and S. Haridi, "Efficient broadcast in structured p2p networks," IPTPS, pp.304–314, 2003.

[5] L. Galanis and D.J. DeWitt, "Scalable distributed aggregate computations through collaboration," DEXA, pp.797–807, 2005.

[6] M. Castro, P. Druschel, A.M. Kermarrec, A. Nandi, A.I.T. Rowstron, and A. Singh, "Splitstream: High-bandwidth content distribution in cooperative environments," IPTPS, pp.292–303, 2003.

[7] Z. Zhang, S. Chen, Y. Ling, and R. Chow, "Capacity-aware multicast algorithms on heterogeneous overlay networks," IEEE TPDS, vol.17, no.2, pp.135–147, 2006.

[8] K. Huang and D. Zhang, "DHT-based lightweight broadcast algorithms in large-scale computing infrastructures," FGCS, vol.26, no.3, pp.291–303, 2010.

[9] M. Cai and K. Hwang, "Distributed aggregation algorithms with load-balancing for scalable grid resource monitoring," IPDPS, pp.1–10, 2007.

[10] J. Li, K.R. Sollins, and D.Y. Lim, "Implementing aggregation and broadcast over distributed hash tables," Computer Communication Review, vol.35, no.1, pp.81–92, 2004.

[11] M. Wählisch, T.C. Schmidt, and G. Wittenburg, "Broadcasting in prefix space: P2P data dissemination with predictable performance," ICIW, pp.74–83, 2009.

[12] P. Yalagandula and M. Dahlin, "A scalable distributed information management system," SIGCOMM, pp.379–390, ACM, 2004.

[13] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for internet applications," IEEE/ACM TON, vol.11, no.1, pp.17–32, 2003.