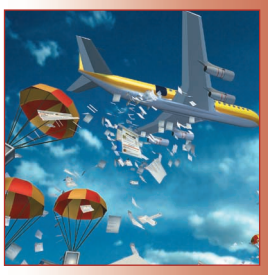

Autonomous Decentralized Community Communication for Information Dissemination

To meet the increasing demand for real-time content delivery, the proposed Autonomous Decentralized Community Communication system offers an efficient information dissemination infrastructure with a decentralized architecture. ADCC's aim is to help end-user communities communicate and exchange information efficiently; to meet this goal, the system uses an application-level multicast technique that arbitrarily scales to large groups. The ADCC system also features a scalable community construction and maintenance scheme that eases the burden of organizing an online community network.

Khaled Ragab, Naohiro Kaji, Yuji Horikoshi, Hisayuki Kuriyama, and Kinji Mori
Tokyo Institute of Technology

Internet services offer rapidly evolving and frequently accessed information spaces for anyone, anywhere, anytime. Service providers build such publish-and-subscribe services from their own viewpoints: they provide the same information to all end users, regardless of demand and situation (such as location, time of day, and so on). This Web-based publish–subscribe scheme has two traditional models. The *pull model* requires users to access to the SP periodically to retrieve new information, while the *push model* requires SPs to deliver content that changes frequently.

Each of these models has disadvantages. With the pull model, users might access information and find that it's unchanged since their last access or that it contains a large, redundant subset of earlier content retrievals. This model is also vulnerable to flash crowds — rapid, sharp surges in requests that overwhelm the server and dramatically increase response time. For its part, the push model fails to take advantage of the Internet's collaborative potential: currently, 90 percent of Internet resources are invisible and untapped.[1] The push model typically uses a one-to-many model in which the SP delivers content directly to each user. Clearly, this approach has scalability limitations.

 1089-7801/04/$20.00 © 2004 IEEE Published by the IEEE Computer Society

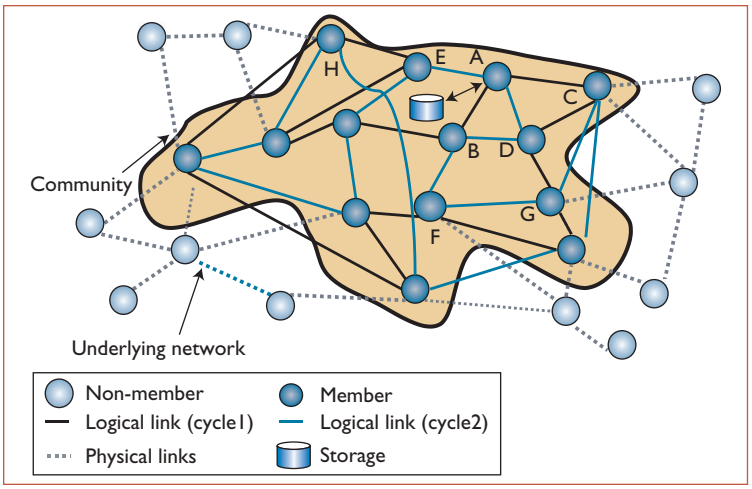

Figure 1. The autonomous decentralized community architecture. Nodes B, C, D, and E are A's immediate neighbors. Unlike peer-to-peer systems, ADCC members cooperate for the good of both individual members and the community as a whole.

To address the drawbacks of the pull and push models, we've designed an information-dissemination infrastructure that offers efficient and cooperative real-time content delivery. Our Autonomous Decentralized Community Communication (ADCC) system was inspired by the autonomous decentralized system (ADS) concept.[2] ADCC is an autonomous decentralized architecture formed by a community of end users (members) with similar interests and demands.[3] The system lets community members maintain autonomy while they cooperate and share information without overloading any single node. The system's autonomous decentralized communication technique offers members flexible cooperation and communication using a hybrid pull–push approach: when community members download interesting content from the server, they forward it to all community members.[4] Here, we describe ADCC and offer results from a simulation of a large, real-world community.

ADCC Overview

ADCC blends the cooperative spirit of social communities with the ADS concept.[5,6] In contrast to the anyone-anywhere-anytime model, ADCC's goal is to provide specific users with information at a specific place and time.

ADCC Community

We define an autonomous community as a coherent group of active members who have individual objectives, common interests, and information needs that occur at a specified time, and sometimes in a specified place[3] (as when an earthquake occurs in Japan, for example, or a particular mall is having a sale at a specific time).[7,8] Community members work together to enhance each member's individual objectives in a timely manner.

Each ADCC member acts as both an information sender and receiver. Furthermore, when a participant sends out information, it is meaningful to all members, and all members want to receive information from all senders in the community. Unlike peer-to-peer (P2P) systems, communication among ADCC members is multilateral: community members cooperate for the satisfaction of individuals and the whole.

Architecture

ADCC's community network is a self-organized logical topology in which a set of nodes, V, considers the symmetric connectivity and existence of loops. Each node tracks its immediate neighbors in a table. Community node X's immediate neighbors are defined as the set of nodes

$$INS_x = \{y; x, y \in V, h(x, y) = 1\} \quad (1)$$

where $h(x, y)$ is the number of logical hops between nodes X and Y.

Each node knows its neighbor nodes and shares this knowledge with other nodes to form a loosely connected network. In Figure 1, the solid lines represent logical links among nodes. Based on its user's preferences, each node judges autonomously whether to join or leave the community network by creating or destroying the logical links with its neighbor members. Users communicate their preferences using a Java utility (such as Java Developer Almanac 1.4), which then generates XML data. The community's boundaries change dynamically along with member requirements. As Figure 1 shows, the ADCC architecture has no central server.

Construction and Maintenance

Our community network will be symmetric in that each node will have identical network duties. As mentioned, there is no central controller for membership management. We have three goals for network construction and maintenance. First, we want to construct a community network that can efficiently support broadcast. We hope to avoid network hotspots by distributing the network traffic evenly among nodes during the broadcast. Second, the network construction must be highly scalable. Finally, the community network's topol-

ogy must provide redundancy. Node failures must not lead to the network disconnecting or severely hampering broadcast properties.

To accomplish these goals, we must build a self-organized community network that lets nodes efficiently join or leave the system, maintains a short network diameter, avoids hotspots, and achieves fault tolerance. Therefore, we've designed the community network as follows.

Community network construction[3] polices the nodes joining and leaving the network and organizes them in a $2d$ regular graph: $G = (V, E)$, in which V is the set of nodes with labels $[M] = \{1, 2, ..., M\}$, and E is the set of edges. Because nodes join and leave frequently, E and V change over time. The graph G is composed of independent d edge-disjoint Hamilton cycles (HCs), which connect all nodes in a graph and visit each node only once.[9] Each node has $2d$ neighbors — establishing node connectivity — labeled as $r_p^{(1)}$, $r_s^{(1)}$, $r_p^{(2)}$, $r_s^{(2)}$, ..., $r_p^{(d)}$, $r_s^{(d)}$. For each i, $r_p^{(i)}$ denotes the neighbor node's predecessor and $r_s^{(i)}$ denotes the neighbor node's successor on the i-th HC. Using HC is advantageous in that nodes joining or leaving require only local changes in the network. Having covered the basics of community, we can now illustrate ADCC's joining, leaving, and fault-tolerance processes.

Joining and Leaving

To join a community, end user A must discover at least one community node, X. To do so, A can either use information from an out-of-band bootstrap mechanism (such as Narada[10]) or employ network broadcasting and discovery techniques such as IP multicast.

Node A sends a `join` request to the newly discovered community node X, which must then find $2d$ neighbors in order for node A to connect. If some joining nodes connect to node X's neighbors, the network diameter increases linearly (as in a completely ordered regular graph).[11] To avoid this, joining nodes in our network select $2d$ neighbors randomly.[3] For our purposes here, however, each node autonomously determines how many new nodes have connected to it within period t; we call this the *node join-rate* $\varphi(t)$.

Node X broadcasts a `join` request to all community nodes within $O(log_{2d} M)$ layers. Each node autonomously decides, based on its join rate $\varphi(t)$, whether to reply "Ok to join." Node X receives some "OK to join" messages, from which it autonomously selects d nodes in different HCs. As Figure 2 shows, the selected d nodes then call the `add()` routine to add joining node A in each i-th HC. The add routine inserts the joining node between the calling node and its successor in the i-th HC. For the edge between the calling node and its successor, the routine substitutes two edges, one between calling and joining nodes and one between the joining node and the calling node's successor. The `Edge(B, C, i)` routine makes C B's successor and B C's predecessor. It thus creates the communication session between nodes B and C. Obviously, the join process requires only local network changes.

When a member leaves the community, it notifies its neighbors and calls the `leave()` routine for each HC. This routine creates edges at d between the leaving node's successor and predecessor nodes. Again, this requires only local network changes with $O(d)$ messages.[3]

Fault Tolerance

To achieve fault tolerance, we assume that each node knows the predecessor of its predecessor node and the successor of its successor node in each cycle.

To detect node failure, the neighboring nodes in the INS_x periodically exchange keep-alive messages with node X. If node X is unresponsive for a period T, neighbor nodes presume it has failed. All neighbors of the failed node update their INS sets and execute the `FT()` routine (see Figure 2), which connects two nodes around the failed node in the same cycle and sets the calling node's predecessor and successor to the failing node's predecessor and successor, respectively. This maintains the HC as well as the same number of links for all nodes.

This technique scales well: a few nodes exchange messages to detect faults, and fault recovery is local, involving only a few nodes $|INS_x|$. In addition, this technique maintains the network G composed of disjoint HCs. If a Hamilton path connects every two nodes of G, then G is Hamilton-connected.[12] Thus, the network is unpartitionable; to prove this, we construct HC with ($M \geq 5$). For example, if we construct the community network as a four-regular, four-connected graph (in which each node has four neighbors), then it should have two edge-disjoint HCs.[13] The community network is thus a connected graph. Because our proposed fault-tolerance technique maintains the HC, the resulting network is connected and unpartitioned.

ADCC Communication

Most Web browsers use a one-to-one communication protocol that gobbles up network bandwidth and leads to unresponsive real-time services. One-

to-many communication models, which feature a single group controller, are also an option, but might not scale well for large groups (thousands of members) or for dynamic multicast groups that have frequent joins and leaves.

Further, while several implementations of the application-level multicast protocol exist,[10,14] none of the current designs scales to large groups. Also, because conventional unicast and multicast technologies use the destination address to send data, they're not applicable in changing environments in which joins and leaves are frequent.

The ADCC communication technique aims to provide

- code content based on community services to provide flexibility, and
- multilateral communication to let members cooperate and attain mutual benefits in a timely and productive manner.

ADCC will thus offer a scalable and flexible alternative to current communication methods.

Content Coding

Our technique first separates the community service's[15] logical identifier from the physical node address. Rather than specifying the destination address, the sender sends a message to neighbor nodes. This message contains three fields. The *content code* expresses the topic of interest; the sender assigns it based on a type of community service, so that the service can act as a logical node. Examples include codes for politics, news, and so on. The *characterized code* uniquely defines further message content details. The final field contains the data (request) itself.

Multilateral Communication

Multilateral communication typically occurs among community members who are already networked on a bilateral basis. In our 1→*N* community communication,[4] the community node asynchronously sends a message to *N* neighbor nodes. The neighbor nodes then forward the message to another *N* nodes in the next layer, except the node that delivered the incoming message; this occurs until all community nodes have received the message.

As Figure 2 shows, when a node joins the community, it executes the `listen()` routine and listens to all messages propagated in the community network that hold the community content code. The message's source calls the `Forward_message()` routine, which sends the message to all its neighbors. As soon as a node receives a message, it also calls the `Forward_message` routine, which checks to see whether the message has been received before. If not, it sends that message to all neighbors except the node that delivered it.

Congestion is possible if some community nodes simultaneously send identical messages. To avoid this, each node remembers the characterized code of recently routed messages, and uses this to decide autonomously whether to forward the received messages to neighboring nodes. Moreover, each node autonomously decides whether to keep or delete the received message's characterized code based on how frequently it receives such messages.

As this description shows, the 1→*N* communication technique does not rely on a central controller. Each node has its own local information and communicates only with *N* neighbor nodes. Unlike in IP multicast group address[16] or multicast service nodes,[17] our method uses no global information.

Communication Protocols

Our community communication technique has two protocols, one based on a *hybrid pull–push* model and the other on a *request–reply-all* model.

In the hybrid pull–push-based protocol, community members publish new information to all members using 1→*N*. A typical application is sharing news information among community members at a specific time or place (here, we refer only to moderately sized nonmultimedia content). The push/pull-based protocol offers an effective solution to the flash crowd problem as follows. When community member S downloads interesting content from the server (such as a news server), she publishes it to all community members, thereby relieving the server of the task and distributing the load among the community nodes. When the number of nodes increases sharply, the load at each node increases slightly. The push/pull-based protocol represents a scalable solution for large-scale information-dissemination systems.

When community members want to locate information, they use the request–reply-all-based protocol to send request messages. Other members then cooperate to locate the requested information. If a node finds no results, it forwards the request to its neighbor nodes using 1→*N*. Otherwise, the node sends its results — such as pointers to information or the actual content, depending on its size — in a

reply message to the requesting node and all community members. The reply-all protocol lets all members expand their knowledge and experience without issuing specific requests. This is in keeping with the community's goal of multilateral benefits. The protocol also decreases the per-node traffic by avoiding multiple requests for the same content.

Contrary to P2P communication techniques, in $1 \rightarrow N$ community communication, all members cooperate on a single request to the benefit of all members. In P2P, peers cooperate only for the unilateral benefit of the requesting member.

Performance

To evaluate ADCC's efficiency compared with conventional communication techniques, we use the following metrics.

- *Latency* measures the communication delay from one community node to all others.
- *Relative mean-delay penalty* defines the ratio of the mean delay between ADCC and unicast community nodes as compared to their IP multicast delay. Essentially, RMDP measures the increased delay that applications perceive while using ADCC.
- *Stress* measures the number of identical message copies that a physical link carries. Obviously, we'd like to keep the link stress on all links as low as possible.

To evaluate performance, we developed a simulation over a randomly generated network with different communication costs between nodes. In addition to demonstrating that the ADCC architecture performs well in realistic Internet settings, our simulation illustrates performance issues related to large communities.

Simulation Setup

We ran a simulation on a network topology with 100 routers connected by core links. We used the Georgia Tech random graph generator[18] to generate the network as a transit-stub model. We assigned a random link delay of 4 to 12 milliseconds to each core link; as computed by the graph generator, the average core link delay is 13 ms. We randomly assigned the community end nodes to core routers, with uniform probability. Each community end node was directly attached to its assigned router by a LAN link. We set each LAN link's delay to 1 ms. End nodes joined the community network at a rate of 100 nodes per second with uniform distribution (that is, one node every 10 ms). The leaving rate was 10 nodes per second

```
Add (A, i)
    {
        Successor_node := (Calling_node r →s(i));
        Edge (Calling_node, A, i);
        Edge (A, Successor_node, i);
    }

Edge (B, C, i)
    {
        ( B →rs(i) ) := C ;
        ( C →rp(i) ) := B ;
    }

Leave ( )
    { // LN is leaving node
        For i :=1, …, d in parallel
            do Edge (LN →rp(i),LN →rs(i), i)
     }
FT(x, i )
    { // x is the failed node in cycle i.
        MyPred := (Calling_node →rp(i));
        MySucc := (Calling_node →rs(i));
        If (x==MyPred) then
           (Calling_node →rp(i)) :=(MyPred → rp(i));
        If (x==MySucc) then
           (Calling_node →rs(i)) := (MySucc → rs(i));
    }

Listen(CC)
    {// CC is the community communication content
          code
    // The self-variable means the calling node id.
        do{
            if (rec_message.cc == CC) then self.
                Forward_message(rec_message);
            } while(1);
    }

Forward_message(m)
    {// m is the message structure contains
        // the CC, CH and data
            If ( self.Not_received_before(m) ) then
               for k ∈ {self. Neighbors} −
                         {self.received_from(m)}
                    in asynchronously do
                             self.SendTo(m, k);
    }
```

Figure 2. The join and leave processes. The expression `(h → Var)` *means that the calling node seeks the value of* `Var` *from node h.*

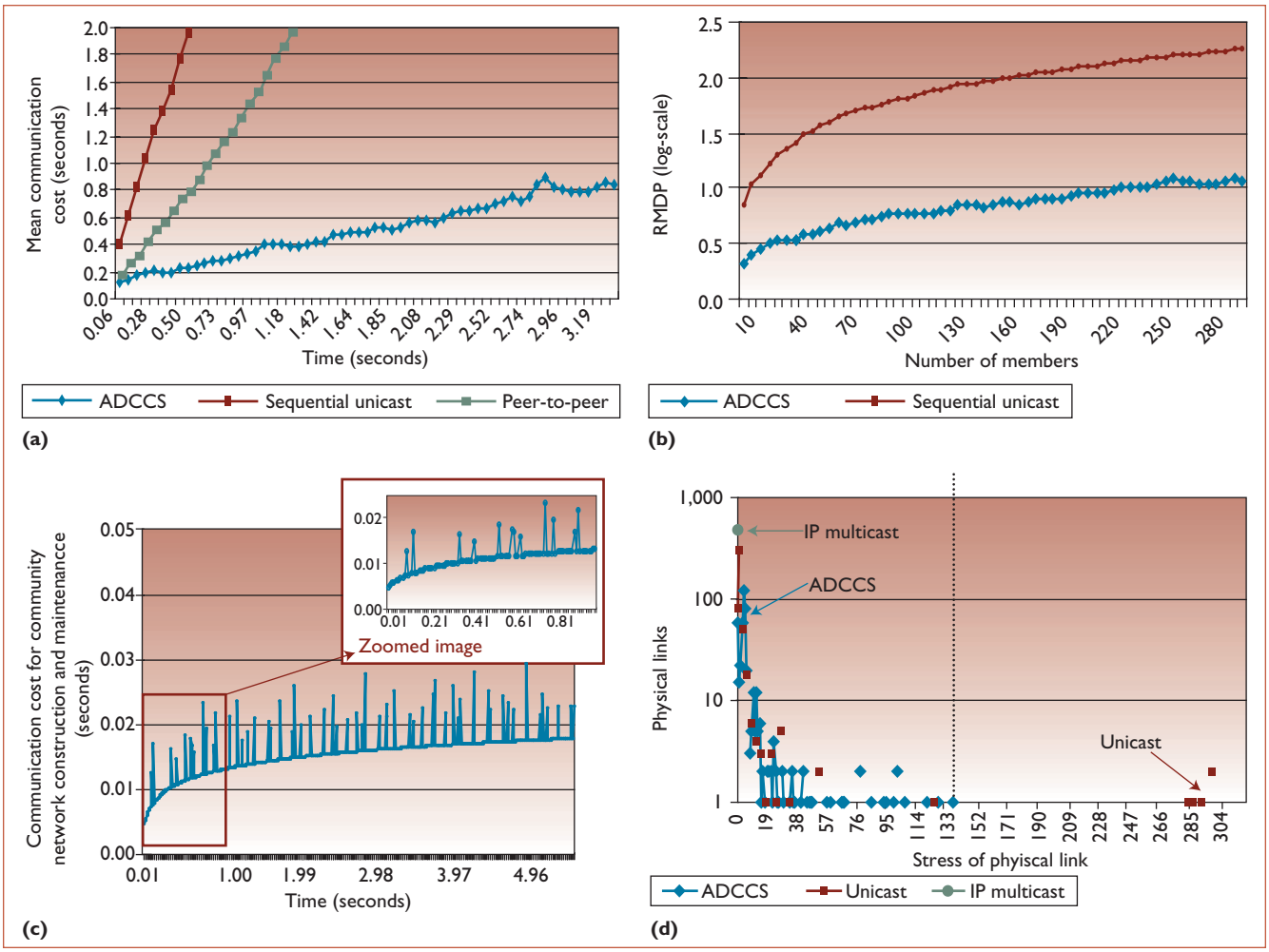

Figure 3. Simulation results. Results for (a) the mean communication cost variations for sending a message from one node to all nodes, and (b) the RMDP variation for ADCC and sequential unicast, for the first 3.5 seconds of simulation. (c) Results for construction and maintenance overhead for 5.5 seconds of simulation with 500 members. (d) The physical link stress variations for ADCC, IP multicast, and naïve unicast.

with random distribution. The community network spends four-array connectivity for each end node.

Communication Results

We conducted a simulation to compare ADCC with unicast and P2P communication technologies. We constructed a P2P overlay network as a completed order-regular graph.[11] The network construction scenario is as follows. When the majority of joining nodes connect to the neighbors of a specific node, the P2P overlay network diameter increases linearly with the increase in network size. In each simulation run, we randomly selected one community member as the source and then evaluated the required communication cost to send a message to all nodes.

For simplicity, Figure 3a shows the results of only the first 3.5 seconds of the simulation's 20-second running time. The figure plots the mean communication cost (MCC) variations required to send a message from one node to all nodes at various points during the experiment. ADCC showed approximately 90-percent MCC improvement compared to unicast. Given ADCC's construction techniques (which yield a short network diameter), ADCC showed about 70-percent improvement compared to P2P. Figure 3b plots the RMDP variation for both sequential unicast and ADCC. Compared to unicast, ADCC showed about a 90-percent RMDP improvement.

Construction and Maintenance Overhead

The cost of joining communication is the time required to forward the join request message within the community network. We ran this simulation for 20 minutes; as a result, the community net-

Related Work in Data Dissemination Infrastructures

During the past decade, much work has demonstrated information dissemination infrastructures' usefulness to large-scale distributed systems.

Like Overcast,[1] Narada,[2] and the Application-Level Multicast Infrastructure (ALMI),[3] the Autonomous Decentralized Community Communication system (ADCC) implements multicast, uses a self-overlay network, and assumes only unicast support from the underlying network layer. Whereas Narada and ALMI are dedicated to collaborative applications used by small groups, however, ADCC is a framework for collaborative applications used by large groups.

Scattercast[4] and Overlay Multicast Network Infrastructure (OMNI)[5] are designed for global content distribution and media-streaming infrastructure support, with proxies deployed across the Internet to support many clients. For large-scale data distributions, such as live Webcasts, they use a single source. In contrast, ADCC nodes are peers that are organized in a community network. The community concept is a "real" end-system multicast approach: The end systems (individual members) work cooperatively to deliver the data to the whole community.

ADCC is dedicated for multisender applications with many participants. It does not depend on the multicast support by routers (as in IP multicast) and does not depend on multicast service nodes (as in Scattercast and OMNI); such a reliance clearly constitutes a single point of failure and entails vulnerability to flash-crowd problems.

ALMI takes a centralized approach to the tree-creation problem, which constitutes a single failure point for all the group's control operations. In contrast, ADCC takes a decentralized approach — no one node knows the total system.[6] Scattercast and Narada take a mesh-based approach, in which every member should maintain a full list of all other members. This approach does not scale well. ADCC scales well to large groups because each member is required to know only neighboring members.

References

1. J. Jannotti et al., "Overcast: Reliable Multicasting with as an Overlay Network," *Proc. Symp. OS Design and Implementation* (OSDI), Usenix Assoc., 2000, pp. 197–212.
2. Y.H. Chu et al., "A Case for End System Multicast," *Proc. Int'l Conf. Measurement and Modeling Computer Systems* (ACM Sigmetrics), ACM Press, 2000, pp. 1–12.
3. D. Pendarakis et al., "ALMI: An Application-Level Multicast Infrastructure," *Proc. Third Usenix Symp. Internet Technologies and Systems* (USITS), Usenix Assoc., 2001, pp. 49–60.
4. Y. Chawathe et al., *Scattercast: An Architecture for Internet Broadcast Distribution as an Infrastructure Service*, doctoral dissertation, Computer Science Dept., Univ. of California, Berkeley, Dec. 2000.
5. S. Banerjee et al., "Construction of an Efficient Overlay Multicast Infrastructure for Real-Time Applications," *Proc. Joint Conf. IEEE Computer and Comm. Societies* (INFOCOM), IEEE Press, 2003, pp. 1–11.
6. K. Ragab et al., "Autonomous Decentralized Community Concept and Architecture for a Complex Adaptive Information System," *Proc. IEEE Workshop Future Trends of Distributed Computer Systems* (FTDCS), IEEE CS Press, 2003, pp. 9–15.

work's size became 108,000 members and the required communication cost for network construction and maintenance was about 384 ms.

In the simulation, we randomly selected 10 nodes to leave the network each second by calling the `leave()` routine; maintenance costs varied according to the cost of communicating to the leaving node's successor and predecessor neighbors. Figure 3c shows partial results for 5.5 seconds of simulation with 500 members. The peaks represent the leaving costs. The results show a logarithmic increase in communication cost for network construction and maintenance, with standard deviation approximately equal to 0.0033. Such results indicate that the network-construction and maintenance techniques are scalable to a large membership.

Link Stress

To evaluate physical link stress, we used a simulated community of 300 members. We randomly selected one member as a source and then evaluated each physical link, studying the variation in link stress for ADCC, IP multicast, and naïve unicast (see Figure 3d). As the figure shows, the stress is at most one message per physical link for IP multicast. As expected, most links endure little stress under both ADCC and naïve unicast. The significance, however, is in the plots' tail: under naïve unicast, we measured stress on a single link at 299, because links near the source are highly stressed. In contrast, ADCC distributes the stress more evenly across the physical links, decreasing the overall link stress more than 50 percent versus naïve unicast.

Conclusion

We're currently extending our work on ADCC in several directions. Because latency between members is an important criterion, we're planning to optimize it by organizing the community as several subcommunities. Each subcommunity will have a leader, and the latency from any member to the leader will be bounded by a specific value. We're now investigating how to construct and maintain these subcommunities. We're also studying ways to enhance communication costs while maintaining acceptable construction and maintenance overhead.

References

1. R. Dornfest, "Dark Matter, Sheep and the Cluster: Resolving Metaphor Collision in P2P," panel discussion, O'Reilly Peer-to-Peer and Web Services Conf., Nov. 2001; an overview is available at http://conferences.oreillynet.com/cs/p2pweb2001/view/e_sess/1612.
2. K. Mori, et. al., "Proposition of Autonomous Decentralization Concept," *J. IEE Japan*, vol. 104, no. 12, 1984, pp. 303–310 (in Japanese).
3. K. Ragab et al., "Autonomous Decentralized Community Concept and Architecture for a Complex Adaptive Information System," *Proc. IEEE Workshop Future Trends of Distributed Computing Systems* (FTDCS), IEEE CS Press, 2003, pp. 9–15.
4. K. Ragab et al., "Scalable Multilateral Communication Technique for Large-Scale Information Systems," *Proc. IEEE Computer Software and Applications Conf.* (COMPSAC), IEEE Press, 2003, pp. 222–227.
5. K. Mori, "Autonomous Decentralized Systems: Concept, Data Field Architecture and Future Trends," *Proc. Int'l Symp. Autonomous Distributed Systems* (ISADS '93), IEEE Press, 1993, pp. 28–34.
6. K. Mori et al., "Autonomous Decentralized Software Structure and its Application," *Proc. IEEE Fall Joint Computer Conf.* (FJCC '86), 1986, pp. 1056–1063.
7. T. Ono et. al., "Autonomous Cooperation Technique to Achieve Fault-Tolerance in Service Oriented Community System," *Proc. Int'l Workshop Autonomous Decentralized Systems*, IEEE Press, 2002, pp. 84–89.
8. N. Kaji et. al., "Autonomous Cooperation Technologies for Achieving Real Time Property and Fault Tolerance in Service Oriented Community System," *Proc. Int'l Workshop on Assurance in Distributed Systems and Networks,* IEEE CS Press, 2003, pp. 36–41.
9. B. Jackson and H. Li, "Hamilton Cycles in 2-Connected k-regular Bipartite Graphs," *J. Combinatorial Theory,* series A, vol. 44, no. 2, 1998, pp. 177–186.
10. Y.H. Chu et al., "A Case for End System Multicast," *Proc. Int'l Conf. Measurement and Modeling Computer Systems* (ACM Sigmetrics), ACM Press, 2000, pp. 1–12.
11. A. Oram, *Peer-to-Peer Harnessing the Power of Disruptive Technologies*, O'Reilly and Assoc., 2001.
12. J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications*, Macmilliam Press, 1976.
13. B. Granbaum et al., "Pairs of Edge-Disjoint Hamiltonian Circuits," *Aequationes Math*, vol. 14, nos. 1–2, 1976, pp. 191–196.
14. J. Jannotti et al., "Overcast: Reliable Multicasting with as an Overlay Network," *Proc. Symp. OS Design and Implementation* (OSDI), Usenix Assoc., 2000, pp. 197–212.
15. T. Ono et. al., "Service-Oriented Communication Technology for Achieving Assurance," *Proc. Int'l Workshop on Assurance in Distributed Systems and Networks,* IEEE CS Press, 2002, pp. 69–74.
16. S. Deering and D. Cheriton, "Multicast Routing in Datagram Internetworks and Extended LANs," *ACM Trans. Computer Systems*, vol. 8, no. 2, May 1990, pp. 85–110.
17. S. Banerjee et al., "Construction of an Efficient Overlay Multicast Infrastructure for Real-Time Applications," *Proc. Joint Conf. IEEE Computer and Comm. Societies* (INFOCOM), IEEE Press, 2003, pp. 1–11.
18. E. Zegura et al., "How to Model an Internetwork," *Proc. Joint Conf. IEEE Computer and Comm. Societies* (INFOCOM), IEEE CS Press, 1996, pp. 594–602.

Khaled Ragab is a doctoral student in the Computer Science Department at the Tokyo Institute of Technology. His research interests include autonomous decentralized systems, cooperative information-dissemination systems, and application-level multicast. He received his BS and MS degrees in computer science from Ain Shams University, Cairo. He is an IEEE student member. Contact him at ragab@mori.cs.titech.ac.jp.

Naohiro Kaji works for the Japanese Ministry of Economy, Trade and Industry. He received BE and MS degrees in information engineering from Tokyo Institute of Technology. His research interests include autonomous decentralized systems and mobile computing. Contact him at kaji-naohiro@meti.go.jp

Yuji Horikoshi is a master's student in the Computer Science Department at Tokyo Institute of Technology. His research interests include autonomous decentralized systems and assurance. He received a BE in electrical and electronic engineering from Tokyo Institute of Technology. Contact him at horikoshi@mori.cs.titech.ac.jp.

Hisayuki Kuriyama is a master's student in the Computer Science Department at Tokyo Institute of Technology. His research interests include autonomous decentralized systems and assurance. He received a BE in electrical and electronic engineering from Tokyo Institute of Technology. Contact him at kuriyama@mori.cs.titech.ac.jp.

Kinji Mori is a professor in the Computer Science Department at the Tokyo Institute of Technology. His research interests include distributed computing, fault-tolerance, and assurance. He proposed the autonomous decentralized system in 1977 and holds 350 patents. He received his BS, MS, and PhD degrees in electrical engineering from Waseda University, Japan. He is a Fellow of the IEEE and a member of the Information Processing Society of Japan (IPSJ) and the Society of Instrument and Control Engineers (SICE). Contact him at mori@cs.titech.ac.jp.