# A Middleware for Implementation and Evaluation of Application Layer Multicast Protocols in Real Environments

Kazushi Ikeda, Thilmee M. Baduge, Takaaki Umedu, Hirozumi Yamaguchi and Teruo Higashino
Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka, Japan
(k-ikeda,thilmee,umedu,h-yamagu,higashino)@ist.osaka-u.ac.jp

## ABSTRACT

In this paper, we propose a new middleware, which supports implementation and evaluation of Application Layer Multicast (ALM in short) protocols in real environments. The middleware provides many functions by which it can largely reduce the time and effort required for ALM protocol prototyping, performance evaluation and tuning. By using those functionalities, we have easily implemented several well-known application layer protocols and successfully performed their performance comparison in real environments. As an example, we have implemented ALMI, NARADA, NICE and OMNI as ALM protocols, and compared their performance on PlanetLab. We believe that this is the first effort made to develop this kind of ALM supporting middleware.

## Categories and Subject Descriptors

C.2.2 [**Network Protocols**]: Applications; D.2.2 [**Design Tools and Techniques**]: Software libraries

## General Terms

Design

## Keywords

Application Layer Multicast, Middleware, Real Environment

## 1. INTRODUCTION

Application Level Multicast (ALM) has attracted a lot of attentions and a number of ALM protocols, each one has very good quality, have been designed for the last decade [1, 2, 3, 4, 5, 6]. However, in contrast to the success of P2P based file distribution and streaming [7], only a few ALM protocols (end host multicast protocols in particular) were actually implemented and operated [8] even though they have a great potential for launching communication or data sharing without relying on any service except the Internet infrastructure. In this paper, we address the deployment issue of ALM protocols. Our approach is to provide a middleware to facilitate to test and validate the operation and performance of a wide variety of ALM protocols in real and common environments.

Some of ALM protocols have been implemented experimentally or practically, and are ready to run on the Internet. In Ref. [2], an application level multicast communication library called ALMI has been implemented in Java. In Yoid (Your Own Internet Distribution) Project [1] wrapper scripts are provided as Yoid Software for Mbone tools such as vic [9]. End System Multicast (ESM) [10] developed by the research group at Carnegie Mellon University provides a native code toolset based on the methodology in Ref. [3]. This tool is famous for its distribution of live video in SIGCOMM2002 conference HyperCast [11] provides socket-like Java APIs based on Delaunay Triangulation methodology in Ref. [12]. RelayCast[13] is a middleware to aim at adapting to various applications that require different metrics. Our research group has also presented Java implementation of EMMA [14].

Different from these toolsets which mostly implemented specific protocols in different languages for experiment purposes, we provide a framework to design and test (existing or new) ALM protocols in *real*, *common* environments. In the sence of toolsets for overlay protocols, MACEDON [15] is a well-designed toolset which supports the development of overlay protocols. However it mainly concentrates on implementing DHT protocols and widely supports P2P overlays while our middleware is specified for the development of ALM protocols and provids more various support functions. Testing ALM protocols in real environments is quite important since their performance is strongly affected by end hosts' capabilities and configurations, while they are highly abstracted or ignored in simulations. Moreover, the common environment enables us to build and run several ALM implementations and compare them. Performance comparison of ALM protocols by simulation has been conducted [16] and dedicated simulators have been provided [17]. However, to our best knowledge, no paper has reported the comparison of ALM protocols in real environments under common settings, and no tools and methodologies for such purpose have been presented so far. Actually, we have implemented the algorithms of ALMI [2], NARADA [3], NICE [18], OMNI [5] and compared their performance on PlanetLab to show that our middleware could help for quick and brief imple-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*NOSSDAV* '07 Urbana, Illinois USA
Copyright 2007 ACM 978-1-59593-746-9/06/2007 ...$5.00.

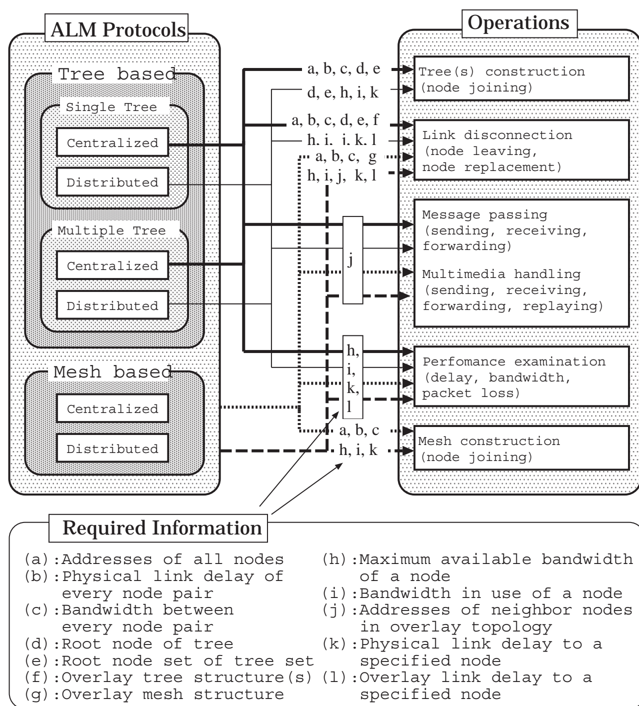


**Figure 1: ALM protocol classification with operations**

mentation of their prototypes and efficient testing in real environments.

# 2. MIDDLEWARE DESIGN AND IMPLEMENTATION

## 2.1 Characterizing ALM applications

Existing ALM protocols can be categorized from the aspects of overlay topology (single tree, multiple tree or mesh) and management policy (centralized or decentralized). This categorization is shown in Figure 1 along with the operations and required information to perform them. After well studying these characteristics, our middleware is designed to give the maximum flexibility and functionality to all kinds of ALM protocol development.

## 2.2 Functionalities of Middleware

Our middleware runs on each participant node. The functionalities of the middleware are categorized to the following six categories.

### 2.2.1 Topology management functions

In most ALM protocols, (1) overlay topological information, (2) available bandwidth of nodes, (3) connectivity and (4) delay between each pair of nodes are utilized. For instance, newly joining nodes are favorably connected close to the source node as to satisfy the delay constraint in a multimedia streaming application. These functions are described in the rest of this section, after categorizing them into, (a) common functions, (b) centralized protocol specific functions and (c) other protocol specific functions.

The common functions provide the following information, which can be used by all protocol types of Figure 1. As *node's information*, node's available bandwidth (dedicated LAN bandwidth), utilized bandwidth ((h) and (i) in Figure 1) and any other protocol-specific information are provided. In addition, IDs and network addresses of neighboring nodes ((j) in Figure 1) are provided as *node's connectivity information*. Finally, overlay and physical network delay to any other node ((k) and (l) in Figure 1) is provided as *delay information*. This information is stored in each node in a decentralized scheme and in the administrative node in a centralized scheme.

The centralized protocol specific functions are common for all types of centralized ALM protocols. Their corresponding information includes: IDs and network addresses of all participant nodes ((a) in Figure 1) as *node set information*, physical network delay and bandwidth between each pair of nodes ((b) and (c) in Figure 1) as *node pair information*, and join, leave timestamps and communication history as *node log information*. This information is stored in the administrative node.

Finally the other protocol specific functions provide the following information for centralized single-tree, multiple-tree and mesh protocols: root node(s) and tree structure(s) information ((d), (e) and (f) in Figure 1) for single-tree and multiple-tree protocols, the maximum delay and hop count of tree and those from the root node, and mesh structure information for mesh protocols ((g) in Figure 1). This information is stored in the administrative node.

### 2.2.2 Basic communication support functions

Our middleware provides the following functions for basic communication purposes; establishing/destroying overlay links between nodes, message passing (send, receive and forward), and task scheduling for specified timestamps.

### 2.2.3 Multimedia support functions

Streaming protocols with their individual implementations cannot be simply compared as the streaming performance largely depends on the implementation method. So this functionality of our middleware largely benefits the ALM application development; *multimedia passing* (send, receive and forward), and *real-time play backing* of the received multimedia stream (under development). Here, multimedia data passing is done using RTP/RTCP.

### 2.2.4 Performance monitoring functions

The following performance monitoring functions become quite important for the maintenance of developed ALM applications: *real-time displaying* of node connectivity, bandwidth, delay, packet-loss and jitter, *real-time identification* of bottleneck nodes, and *logging* of messages, multimedia passing history and overlay link establishing/destroying history. The information required to realize these functions is acquired from the topology management functions, the basic communication support functions and multimedia support functions.

### 2.2.5 Execution support on PlanetLab

This functionality largely reduces the efforts required so far for carrying out large scale experiments on PlanetLab; *remote manipulation* of nodes such as adding, removing and replacing, and *node task (join, leave, etc) scheduling*.

### 2.2.6 ALM protocol design support functions

These functions are provided for developers as high-level (algorithm level) APIs: (i) *basic tree construction*: each node is greedily connected to the tree satisfying a given condition (e.g. minimum delay), (ii) *k-connected mesh construction*: $k$ neighbors are randomly selected, (iii) *basic tree recovering*: recovering of the tree in a single node absence (leaving or failure), by connecting the disconnected subtree to left-node's parent, (iv) *information collection*: periodic collection of information, such as neighbor node information, tree/mesh diameter or depth and (v) *node swapping*: child-parent, child-grand parent, sibling wise, child wise, random swapping of nodes satisfying a protocol-specified condition (*e.g.* delay).

## 3. IMPLEMENTATION OF ALM PROTOCOLS

In this section, we explain implementation of some well-known ALM protocols, using our middleware. The complete list of classes and APIs of our middleware will be given in [19]. In each of the following examples, we first outline their protocol behavior and then describe how our middleware helps the implementation of those protocols.

### 3.1 ALMI

ALMI is a centralized, tree-based protocol. It mainly consists of tree construction procedure and tree refining procedure. As for the joining procedure new nodes are required to inform their joining ambition to the tree-controlling node. Then the tree-controlling node decides and informs appropriate positions for those nodes. Finally, the new nodes have the responsibility of positioning themselves following the controlling node's instructions[1]. Our middleware implements this join procedure as shown in the pseudo-code of Figure 2 . Note that the code only outlines the behavior of the protocol, omitting the verbose parts like error handling.

A newly joining node sends *ALMIJoinRequestPacket* to the *SuperNode* that controls the entire tree. The middleware provides a handler called *onReceiveALMPacket*, which is executed on receiving a message at a node. Therefore the behavior of SuperNode for ALMIJoinRequestPacket can be described in onReceiveALMPacket handler (line 02), such as, (a) finding the connection situation of the tree by calling *CentralizedTreeConnection* (line 03), (b) getting the list of nodes where the residual degree is not zero (line 04), (c) randomly selecting one among them, and, (d) informing the selected candidate to the joining node by sending *ALMIJoinReplyPacket*.

In ALMI, periodical tree-refining procedures are carried out at SuperNode after the initial construction. The middleware supports this by providing *onTopologyUpdate* handler, which is periodically executed. The actual refining is done by making the nodes change their parents, and our middleware's *ChangeParentPacket* can be used to inform the new parent candidates to the corresponding nodes (lines 16 and 20). Basically, the grand parent is selected as the new parent candidate, and the node with the minimum delay is used in case the grand parent has no residual degree (lines 15 and 18).

[1]Note that these basic steps are typical for most centralized join procedures.

```
01:public void onReceiveALMPacket(
                    ALMNodeId senderNodeId, ALMPacket packet) {
02: if(packet instanceof ALMIJoinRequestPacket){
03:  CentralizedTreeConnection connection = node.
                         getSuperNodeTopology().getConnection();
04:  Vector<ALMNodeId> vec = connection.
                                 getNodeIdsWithResidualDegree();
05:  if(vec.size() > 0){
06:   node.sendPacket(senderNodeId,new ALMIJoinReplyPacket(
         vec.elementAt((int) (Math.random() * vec.size()))));
07:  }
08: }
19:}
10:void onTopologyUpdate(){
11: Vector<ALMNodeId> sessionNodeIds =
                               treeConnection.getKnownNodeIds();
12: for(int i = 0;i < sessionNodeIds.size();i++){
13:  ALMNodeId targetNodeId = sessionNodeIds.elementAt(i);
14:  ALMNodeId grandParentNodeId = treeConnection.
                             getGrandParentNodeId(targetNodeId);
15:  if(treeDegree.hasResidualDegree(grandParentNodeId)){
16:   node.sendPacket(senderNodeId,
                     new ChangeParentPacket(grandParentNodeId);
17:  }else{
18:   ALMNodeId minLatencyNode = treeLatency.
                               getMinLatencyNode(targetNodeId);
19:   if(!treeConnection.
                isParentNodeId(targetNodeId,minLatencyNode)){
20:    node.sendPacket(senderNodeId,
                        new ChangeParentPacket(minLatencyNode);
21:   }
22:  }
23: }
24:}
```

**Figure 2: main part in source code of ALMI**

### 3.2 OMNI

OMNI is also a tree based ALM protocol. But it is different from ALMI in the sense of distributed tree management. However, considering the fact that onReceiveALMPacket handler is waiting at each node, join requests from multiple nodes can be handled simultaneously at different nodes. So the behavior on receiving *OMNIJoinRequestPacket* at each node can be described in onReceiveALMPacket handler (lines 02–03 in Figure 3). Here, the joining is permitted by replying the join requesting node $N_{new}$ with *OMNIJoinReplyPacket*, in case the adopting node has residual degrees (lines 03–05). Otherwise, delays to the current neighbors are measured (line 08) after queuing $N_{new}$ (line 07), and the reply to $N_{new}$ is done depending on these delay values.

Note that it takes different measuring-times because of different network connectivity, and the reply should be done once the delay values of all neighbors are collected. For this purpose, our middleware provides another event handler called *onEvent* which is executed when the delay measurement is complete (lines 12–13). Therefore, the behavior for the corresponding event occurrence can be described in onEvent handler as follows. OMNIJoinReplyPacket is sent to $N_{new}$ (line 21) if it has a smaller delay than $N_{max}$, which is the neighbor with the maximum delay (line 18,19). In the same time *OMNIChangeParentPacket* is sent to $N_{max}$ (line 20). These two messages make $N_{new}$ replace $N_{max}$, and $N_{max}$ move to $N_{new}$'s child position. If $N_{new}$ has no smaller delay than those of the current neighbors, the connection is refused by an *OMNIJoinRejectPacket* (line 23).

This example illustrates how easily our middleware can describe the behaviors of ALM protocols, even though they consist of asynchronous procedures specific for decentralized

```
01:public void onReceiveALMPacket(
                    ALMNodeId senderNodeId, ALMPacket packet) {
02: if(packet instanceof OMNIJoinRequestPacket){
03:  OMNIJoinRequestPacket oPacket =
                                  (OMNIJoinRequestPacket)packet;
04:  if(node.getTopology().getDegree().hasResidualDegree()){
05:   node.getSocketHandler().sendPacket(oPacket.requestNodeId,
                  new OMNIJoinReplyPacket(node.getMyNodeID()));
06:  }else{
07:   joinWaitNodeQueue.add(oPacket.requestNodeId);
08:   node.getTopology().measureLatency(oPacket.requestNodeId);
09:  }
10: }
11:}
12:public void onEvent(Event event) {
13: if(event instanceof LatencyMeasuredEvent){
14:  Vector<ALMNodeId> vec =
         ((LatencyMeasuredEvent)event).getUpdateNodeIds();
15:  for(int i = 0;i<vec.size();i++){
16:   ALMNodeId waitingNode = vec.elementAt(i);
17:   if(joinWaitNodeQueue.contains(waitingNode)){
18:    ALMNodeId maxLatencyNode =
        node.getTopology().getMaxLatencyNode(
                    node.getTopology().getChildrenNodeIds());
19:    if(node.getTopology().getLatency(maxLatencyNode) >
                 node.getTopology().getLatency(waitingNode)){
20:     node.getSocketHandler().sendPacket(maxLatencyNode,
                    new OMNIChangeParentPacket(waitingNode));
21:     node.getSocketHandler().sendPacket(waitingNode,
                  new OMNIJoinReplyPacket(node.getMyNodeID()));
22:    }else{
23:     node.getSocketHandler().sendPacket(waitingNode,
                     new OMNIJoinRejectPacket(maxLatencyNode));
24:    }
25:    joinWaitNodeQueue.remove(waitingNode);
26:    break;
27:   }
28:  }
29: }
30:}
```

**Figure 3: main part in source code of OMNI**

```
01:public void onTopologyUpdate() {
02: if(object instanceof MeshUpdate){
03:  node.getSocketHandler().sendPacket(
      getMulticastNodeIds(),new NaradaUpdatePacket(
           new ConnectionInformation(),node.getMyNodeID()));
04:  topology.getLatency().
                  measureLatency(getRandomKnownNodeIds());
05:  }
06:}
07:public void onEvent(Event event) {
08: if(event instanceof LatencyMeasuredEvent){
09:  Vector<ALMNodeId> updateNodeId =
           ((LatencyMeasuredEvent)event).getUpdateNodeIds();
10:  ALMNodeId connectNodeId = topology.
               getLatency().getMinLatencyNode(updateNodeId);
11:  ALMNodeId secondNodeId = topology.getLatency().
                   getSecondMinLatencyNode(updateNodeId);
12:  if(!topology.getConnection().isConnect(connectNodeId)){
13:   node.getSocketHandler().connect(connectNodeId);
14:  }else if(!topology.getConnection().
                                  isConnect(secondNodeId)){
15:   node.getSocketHandler().connect(secondNodeId);
16:  }
17: }
18:}
```

**Figure 4: update process of Narada**

```
public class LayeredDistributedTopology extends
                                 BasicDistributedTopology{
 private ALMNodeId leaderNodeId;
 private ALMNodeId nextLeaderNodeId;
 private Vector<ALMNodeId> nextLayerNodeIds;
          :
}
```

**Figure 5: update process of Nice**

schemes.

## 3.3 NARADA

NARADA is a mesh-based decentralized protocol. Here, the newly joining node $i$ becomes a member of the mesh by connecting itself to the node sub-set $N = \{n_1, ..., n_k\}$. The entire node list is assumed to be known by some application specific method. We omit the explanation of this joining part, as it is quite similar to that of ALMI or OMNI.

Similar to ALMI[2], NARADA also consists of a periodical refine procedure. However this differs from that of ALMI in the sense of decentralized execution. In NARADA, the delays to non-neighbor nodes are periodically measured and new connections are established to those with less delay. Remembering the decentralized join procedure of OMNI, it is clear that this delay measurement and corresponding replacement tasks can be described in onEvent handler (lines 07–15 in Figure 4). So this decentralized refining procedure can be easily implemented combining onEvent and onTopologyUpdate (lines 03–04) handlers.

## 3.4 NICE

NICE is also a mesh-based decentralized protocol. However it differs from NARADA in the sense of its hierarchical structure. NICE constructs multiple clusters where each one forms a small mesh, and it constructs another overlay mesh connecting those clusters to form a hierarchical mesh. As we already mentioned, our middleware provides the basic classes, BasicCentralizedTopology, BasicDistributedTopology, BasicCentralizedTreeTopology and BasicDistributedTreeTopology, to support centralized / decentralized, tree-based/mesh-based topologies. However, in hierarchical-decentralized mesh topologies like NICE, the management of each cluster and their upper layer mesh, should be done separately.

For this purpose the middleware provides a new class called *LayeredDistributedTopology* extending BasicDistributedTopology, the basic distributed mesh-topology management class. As the basic mesh-topology can be controlled by our middleware's BasicDistributedTopology class, (1) leaderNodeId: the leader node of cluster, (2) nextLeaderNodeIds: the leader of neighboring clusters, (3) nextLayerNodeIds: nodes of lower layer clusters, and, (4) methods to set/get the values of (1)–(3) variables, are required to be added. The code for (1)–(3) is shown in Figure 5. The code for (4) is omitted due to the space limitation.

Figure 6 summarizes the lines of code (LOC) counts for each protocol explained above. All protocols are easily implemented in a few hundred LOCs. This result shows the usability of the middleware. It reduces the work of implementing basic communication functions and multimedia functions. In addition, the functions specialized for ALM protocols largely help the implementation.

[2]Though OMNI also has a refine procedure, we omitted it due to the space limitation.

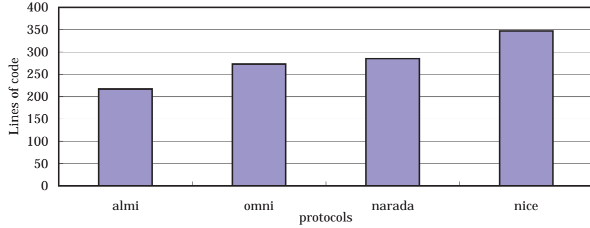


**Figure 6: Number of Lines of source code for each protocol**

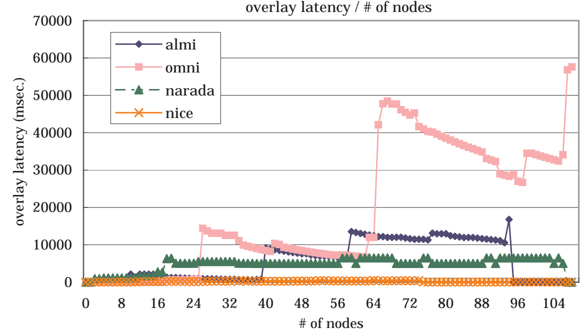


**Figure 7: Delay from source vs number of nodes**

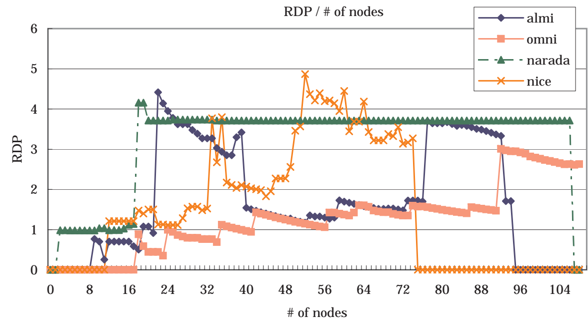


**Figure 8: RDP from source vs number of nodes**

# 4. PERFORMANCE EVALUATION

We compared the performance of ALMI, OMNI, NARADA and NICE, the ALM schemes implemented with the middleware. The experiments were carried out on PlanetLab using our middleware's execution support function, and the multimedia support and performance monitoring functions were used for the evaluation.

## 4.1 Experiment environment

First, we describe our experimental environment.

- number of terminals : about 100 (Asia:10, Europe:40, America:45 and some others)
- terminal configuration : Pentium3 (1.2GHz) – Pentium4 (3.4GHz), 512MB – 3.6GB memory, Linux OS version 2.6.12-1.1398_FC4.5.planetlab
- Java : JDK1.5
- RTT (round trip time) of each pair of PlanetLab nodes : mean 150ms, max 14000ms

## 4.2 Scenario of experiments

Overlay delay and RDP (Relative Delay Penalty) were selected as general evaluation metrics and jitter and streaming bandwidth were selected as the streaming evaluation metrics.

The scenario of our experiments is as follows. First, 20 nodes joined the application forming the initial topology. Then streaming was started after assigning the "center node" as the streaming-source. Here, the node joined first was considered as the center node and the streaming-rate was set to 500kbps, where we assume a video streaming application. Note that we selected the same node as the source node in each protocol.

Next, another 20 nodes were added and the same experiment was carried out, and this was repeated until the number of nodes reached 100.

## 4.3 Experimental results

The experimental results are shown in Figures 7, 8, 9 and 10. Figure 7 shows the average streaming delay (network delay + node delay) of each ALM scheme, and it states that NARADA and NICE, the mesh-based protocols, have smaller delay than ALMI and OMNI, the tree-based protocols. Especially, OMNI's delay is much larger than the others. One possible reason for this is OMNI's defect of performing its refinement only looking at the network delay. So it may select nodes with small link delay and large node delay, which results in high streaming delay. However the performance of these schemes from the aspect of link delay is presented with RDP metric in Figure 8. We can say that the overlay delay is rather large in NARADA. Note that we defined RDP as the ratio of ALM protocol delay (including data buffering delay at intermediate end hosts) to the unicast delay which was measured by ICMP echo packets. Figure 9 also says that the mesh protocols have smaller jitters, which leads to better streaming performance. On the other hand, Figure 10 states that the mesh protocols have bigger bandwidth consumptions.

As it is already clear to the readers from the above experiments, our middleware well supports the performance comparison of variety of ALM protocols. Also as the environment-dependent (thus unexpected) behavior of protocols, we have experienced large jitters in OMNI and ALMI from around 80 nodes. The protocol developers can improve, customize and tune their implementation in real environments by considering those protocols' strong and week features known by their nature of design or by simulations as well as site-specific and application-specific behavior observed using our middleware.

# 5. CONCLUSION

In this paper, we proposed a new middleware for supporting design and implementation of ALM protocols in real environments. Here, we have implemented ALMI, NARADA, NICE and OMNI using our middleware, and carried out

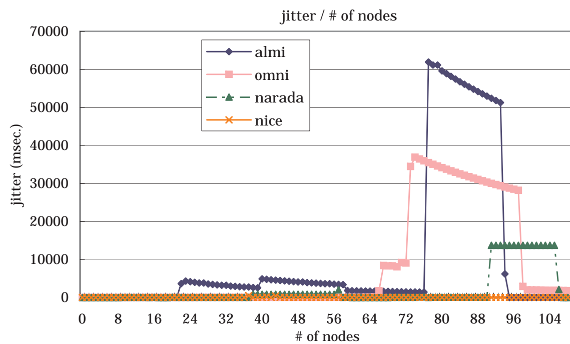


**Figure 9: Jitter vs number of nodes**

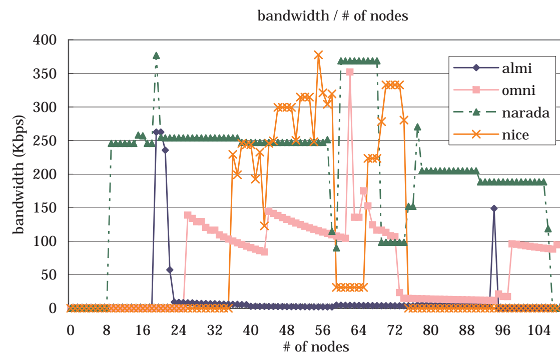


**Figure 10: Bandwidth vs number of nodes**

their performance comparison on PlanetLab. The comparison can be done in a few days, and several comparison results have been obtained using our middleware (we omitted some details of the comparison due to the space limitation). We have a plan to open our middleware from on WWW soon [19]. More detailed comparison with several ALM protocols in real environments is part of our future work.

# 6. REFERENCES


[1] P. Francis, "Yoid: Extending the internet multicast architecture," http://www.isi.edu/div7/yoid/.

[2] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel, "ALMI: An application level multicast infrastructure," in *Proc. of 3rd Usenix Symp. on Internet Technologies and Systems (USITS '01)*, 2001, pp. 49–60.

[3] Y.-H. Chu, S. G. Rao, S. Seshan, and H. Zhang, "A case for end system multicast," *IEEE Journal on Selected Areas in Communication*, vol. 20, no. 8, pp. 1456–1471, 2002.

[4] S. Shi and J. Turner, "Multicast routing and bandwidth dimensioning in overlay networks," *IEEE Journal on Selected Areas in Communications*, pp. 1444– 1455, 2002.

[5] S. Banerjee, C. Kommareddy, K. Kar, S. Bhattacharjee, and S. Khuller, "Construction of an efficient overlay multicast infrastructure for real-time applications," in *Proc. of IEEE INFOCOM 2003*, 2003, pp. 1521–1531.

[6] A. Garyfalos and K. Almeroth, "A flexible overlay architecture for Mobile IPv6 multicast," *Journal on Selected Areas in Communications*, vol. 23, no. 11, pp. 2194–2205, 2005.

[7] Y. Cui, B. Li, and K. Nahrstedt, "oStream: Asynchronous streaming multicast in application-layer overlay networks," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 91–106, 2004.

[8] J. Touch, "Dynamic Internet overlay deployment and management using the X-Bone," *Computer Networks*, vol. 36, no. 2/3, pp. 117–135, 2001.

[9] S. McCanne and V. Jacobson, "Vic: A flexible framework for packet video," in *Proc. of the ACM Multimedia 1995*, 1995, pp. 511–522, http://ee.lbl.gov/vic/.

[10] Y.-H. Chu, S. G. Rao, S. Seshan, and H. Zhang, "Enabling conferencing applications on the internet using an overlay multicast architecture," in *Proc. of ACM SIGCOMM 2001*, 2001, pp. 55–67.

[11] J. Liebeherr and T. K. Beam, "HyperCast: A protocol for maintaining multicast group members in a logical hypercube topology," in *Proc. of 1st Int. Workshop on Networked Group Communication (NGC '99) (LNCS 1736)*, 1999, pp. 72–89.

[12] J. Liebeherr and M. Nahas, "Application-layer multicast with delaunay triangulations," in *Proc. of IEEE Global Telecommunications Conference (Globecom 2001)*, 2001, pp. 1651–1655.

[13] N. Mimura, K. Nakauchi, H. Morikawa, and T. Aoyama, "RelayCast: A middleware for application-level multicast services," in *Proc. of 3rd Int. Workshop on Global and Peer-to-Peer Computing on Large Scale Distributed Systems (GP2PC 2003)*, 2003, pp. 434–441.

[14] T. Yamashita, H. Yamaguchi, K. Yasumoto, T. Higashino, and K. Taniguchi, "Emma Middleware: An application-level multicast infrastructure for multi-party video communication," in *Proc. of 15th Int. Conf. on Parallel and Distributed Computing and Systems (PDCS 2003)*, 2003, pp. 416–421.

[15] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat, "Macedon: Methodology for automatically creating, evaluating, and designing overlay networks," in *Proc. of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, 2004, pp. 267–280.

[16] S. Fahmy and M. Kwon, "Characterizing overlay multicast networks," in *Proc. of IEEE Int. Conf. on Network Protocols (ICNP2003)*, 2003, pp. 61–70.

[17] S. Banerjee, "myns (P2P) simulator," http://www.cs.umd.edu/~suman/research/myns/index.html.

[18] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable application layer multicast," in *Proc. of ACM SIGCOMM 2002*, 2002, pp. 205–217.

[19] "A middleware for implementation and evaluation of application layer multicast protocols in real environments - the APIs," http://www-higashi.ist.osaka-u.ac.jp/software/ALM/middlewareAPI/.