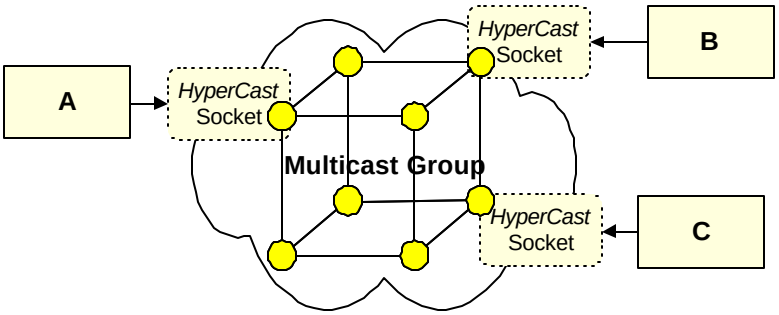
HyperCast: A Super-Scalable Many-to-Many Multicast Protocol for Distributed Internet Applications



A Thesis
In TCC 402
Presented to
The Faculty of the
School of Engineering and Applied Science
University of Virginia
In Partial Fulfillment
Of the Requirements for the Degree
Bachelor of Science in Computer Science

by

Konrad Lorincz

October 1, 2001

On my honor as a University student, on this assignment I have neither given nor received unauthorized aid as defined by the Honor Guidelines for Papers in TCC Courses.

_____

Approved _____ Date _____
Technical Advisor - Jörg Liebeherr

Approved _____ Date _____
TCC Advisor - Rosanne L. Welker

Table of Contents

Table of Figures

Glossary of Terms

broadcast transmission – sending a message to all hosts

heartbeat – time interval at which nodes communicate with each other. The default value is 2 seconds.

IP Multicast – a multicast protocol which builds a virtual network on top of the existing physical network, developed by Steve Deering in the late 1980s

MBONE – Multicast Backbone. A collection of multicast capable routers on the Internet backbone, which forward IP Multicast messages

multicast transmission – sending a message to a subgroup of hosts

overlay protocol – a protocol responsible for constructing and maintaining a multicast group

overlay topology – the topology created by an overlay protocol

router – a network node which takes messages from one network and forwards them onto a different network

unicast transmission – sending a message to a single destination host

Abstract

Unicast transmission is appropriate for many Internet applications such as e-mail, web browsing, and file transfer because these applications involve interaction between a single sender and a single receiver. However, other classes of applications, which involve a sender transmitting the same data to multiple receivers, require a different transmission paradigm. Multicast protocols address this class of applications by efficiently distributing data from a single sender to multiple receivers. However, existing reliable multicast protocols cannot efficiently handle multicast groups with many senders. To address this problem, a novel protocol called *HyperCast* was developed, which efficiently handles multicast groups with many senders. This paper describes the redesign of *HyperCast* and its overlay protocol structure. Large-scale experiments on two overlay protocols *Hypercube* and *DelaunayTriangulation* verify that they scale to multicast group sizes of 10,000 nodes.

Chapter 1: Introduction

Collaborative Internet applications, such as videoconferencing, shared document editors, shared file distributors, and distributed search engines, must send data to more than one destination host [5]. Efficiently distributing data in such applications requires members to join multicast groups for communication. Multicasting allows the sending hosts to deliver data efficiently to a set of receivers [14]. Supporting large multicast groups requires an efficient way to exchange control information between group members in order to construct and maintain the multicast group, and distribute application data to the multicast group members. This project redesigned a new multicast protocol *HyperCast* and verified through experiments that it scales to 10,000 nodes on the Internet.

1.1 Ways of Transmitting Messages in a Network

There are three ways in which messages can be transmitted in a network. They are unicast, multicast, and broadcast transmission. In unicast transmission the sending host sends a message to a single receiving host [14]. The sender must know the unique address of the receiver. If the sender does not know the address of the receiver, then the message cannot be sent. Most Internet applications like e-mail, File Transfer Protocol (FTP), and Web browsing rely on unicast transmission.

In multicast transmission the sending host sends a message to a subset of all the hosts in the network [14]. The subset or group of receivers is identified by a unique multicast group address, which the sender must know. The underlying multicast protocol is

responsible for delivering the message to all multicast group members, and therefore the sender does not have to know the address of all the individual receivers. Internet applications like videoconferencing, distributed white boards, and distributed search engines rely on multicast transmission.

In broadcast transmission the sending host sends a message to all hosts in the network [14]. The sender sends the message to a well-known broadcast address. All hosts on the network listen to this broadcast address and pick up any messages. As in multicast transmission, the sender does not have to know the addresses of any of the hosts in the network. In fact broadcasting is used mostly to discover other hosts and their addresses in a network.

The three different ways of transmitting messages in a network are illustrated in Figure 1-1. Unicast transmission is depicted in Figure 1-1(a) by the single arrow from the sender to one receiver. Figure 1-1(b) depicts multicast transmission by extending several arrows from the sender to a subset of hosts. Broadcast transmission is represented in Figure 1-1(c) by extending arrows from the sender to all hosts in the network.

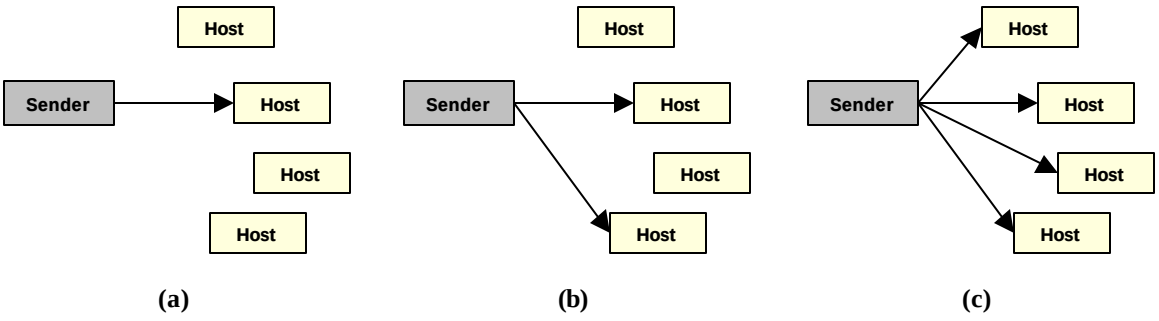


Figure 1-1: Three ways of transmitting messages: unicast (a), multicast (b), broadcast (c)

1.2 Advantages of Using Multicast Protocols

The Internet, a packet switched network, is designed for unicast transmission, the delivery of data packets from a single sender to a single receiver. Unicast is appropriate for the majority of Internet applications such as e-mail, web browsing, and file transfer because these applications involve the interaction between a single sender and a single receiver [9]. However, for other classes of applications, which involve a sender transmitting the same data to multiple receivers, unicast is inefficient because unicast traffic increases proportionally with the number of receivers.

The problem with unicast transmission is that the sender has to send a copy of the message transmitted, to each receiver. As shown in Figure 1-2, although the path from the sender to the receivers is partially shared, multiple copies of the same message must be sent over the shared path. Multicast transmissions take a more scalable approach by addressing a set of receivers rather than a single destination [1]. In Figure 1-3, the sender transmits only one copy of the message over the shared path and lets the routers make multiple copies of the message when the paths to the receivers divide. Multicast messages are therefore addressed to a multicast group as a whole.

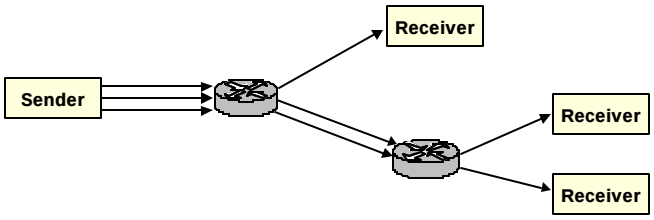


Figure 1-2: Unicast transmission

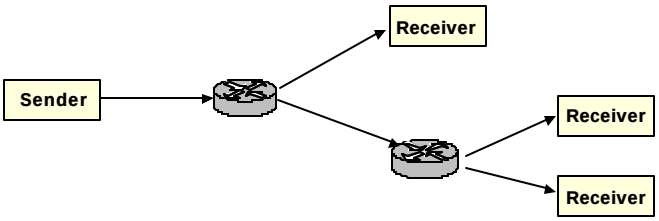


Figure 1-3: Multicast transmission

Multicasting emerged on the Internet with the development of the IP Multicast protocol by Steve Deering in the late 1980s, and the Internet Multicast Backbone (MBONE) [9]. IP Multicast constructs a virtual path in a multicast group from the sender to the receivers, along which the sender transmits messages. The MBONE is a collection of multicast capable routers on the Internet backbone, which forward IP Multicast messages.

1.3 Problems With Existing Multicast Protocols

Multicast protocols can be broken down into two categories: *one-to-many* and *many-to-many*. In *one-to-many* multicast protocols, a single or a few group members (hosts) disseminate messages in a multicast group. In *many-to-many* multicast protocols, most or all group members (hosts) can disseminate messages in the multicast group. Most existing reliable multicast protocols efficiently address the case in which only one or a few members distribute data in a multicast group [4]. *Many-to-many* reliable multicast protocols have problems handling large multicast groups and therefore do not scale well to group sizes of more than a few hundred members.

The major problem with scalable multicast applications is the need to exchange control information between multicast group members in order to maintain the multicast group [3]. Since the control messages need to be reliable, acknowledgements (ACKS) for messages received or negative acknowledgements (NACKS) for messages missing must be sent from the receivers to the sender. If each sender and receiver in a multicast group directly exchanges messages to ensure reliability, than the ACKS or NACKS returning to the sender are proportional to the number of group members. For large groups, this implosion overwhelms the sender and therefore it is not scalable. This problem is known as the ACK/NACK implosion problem.

Several protocols have been proposed to solve the ACK/NACK implosion problem, most of which rely on organizing group members in a tree [1]. In this topology the sending node is the root of the tree. Messages are sent down the tree from each parent to its children. ACKS or NACKS are then sent up the tree from children to their parent. Each receiving parent combines the received ACKS or NACKS from its children into one packet that it sends up the tree to its parent. This process is continued until the original sender is reached. The acknowledgement tree method is scalable because the number of ACKS or NACKS received is proportional to the number of children. As long as well-balanced trees – trees in which the end nodes are at about the same distance from the root – can be constructed, the ACK/NACK implosion problem can be avoided.

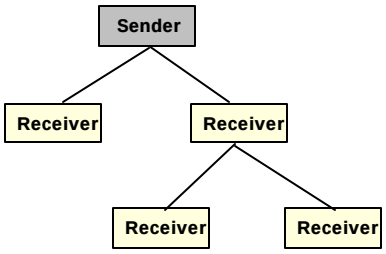


Figure 1-4: Well-balanced acknowledgement tree in which the sender is the root of the tree.

Tree topologies are appropriate for *one-to-many* sender applications because a well-balanced tree can be constructed such that the sending node is the root of the tree. However, this topology is inefficient for groups with many senders. If a node other than the root wants to send data, then a new tree must be constructed or the existing tree re-hung. Constructing a new tree is very costly in terms of time and is therefore not a good solution. Re-hanging the tree incurs little overhead however, the resulting tree may not be well-balanced and therefore it can suffer from a long delay [4.1].

Liebeherr and Sethi address this problem, with the overlay protocol named *Hypercube*, by arranging multicast group members in a logical N-dimensional hypercube on top of a physical topology [4;17]. By exploring the symmetric properties of the hypercube, well-balanced acknowledgement trees can be quickly constructed for any node. The ability to construct well-balanced acknowledgement tree with little overhead makes the *Hypercube* protocol scalable. Figure 1-5 illustrates that *Hypercube* takes group members from a physical topology and organizes them in a logical hypercube topology.

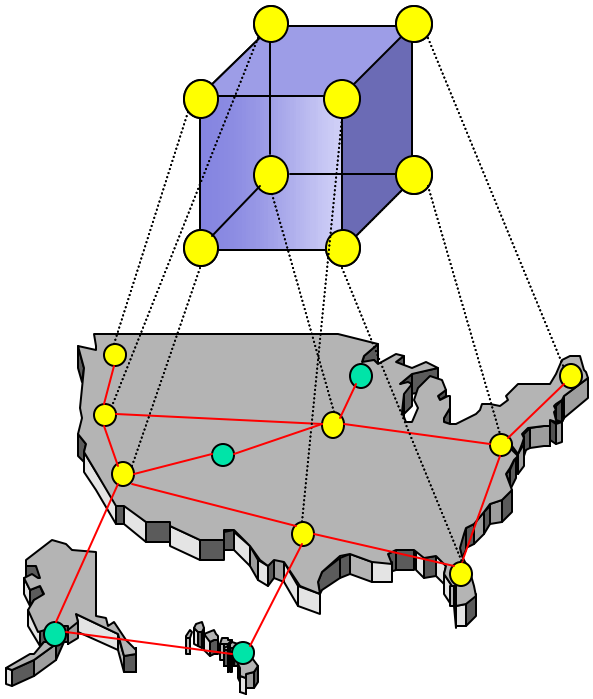


Figure 1-5: Hypercube, Arranges nodes in a logical N-dimensional hypercube on top of a physical topology.

A hypercube is an N-dimensional structure with 2^N nodes, $N*2^{N-1}$ edges and each node has N neighbors [4, 12]. Figure 1-6 shows examples of 1, 2, and 3 dimensional hypercubes. As Figure 1-6 depicts, the 1-dimensional hypercube has $2^1=2$ nodes, $1*2^{1-1}=1$ edges and each node has 1 neighbor. The 2-dimensional hypercube has $2^2=4$ nodes, $2*2^{2-1}=4$ edges and each node has 2 neighbors. The 3-dimensional hypercube has $2^3=8$ nodes, $3*2^{3-1}=12$ edges and each node has 3 neighbors.

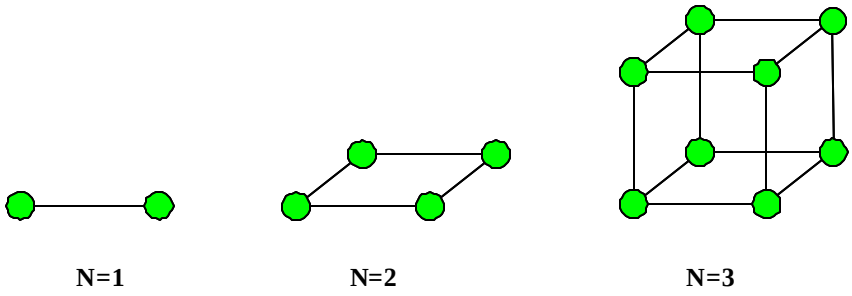


Figure 1-6: Depicts N-dimensional Hypercubes [17]

Building message transmission trees from any node in *Hypercube* is very easy. Figure 1-7 and Figure 1-8 show transmission trees built from two separate nodes. Nodes in a hypercube have a physical address, the IP address of the host, and a logical address used by the logical topology.

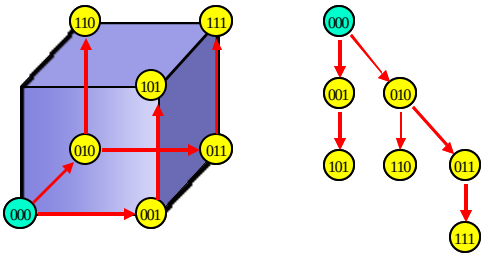


Figure 1-7: Transmission tree with root 000 [17]

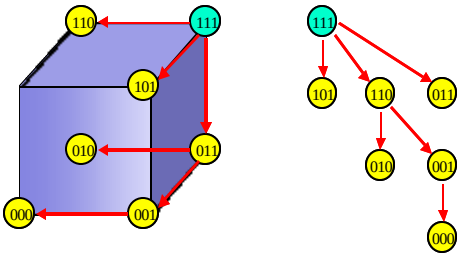


Figure 1-8: Transmission tree with root 111 [17]

1.4 Previous Work

The *HyperCast* protocol designed by J. Liebeherr and B. Sethi was first implemented by Tyler Beam in his masters thesis [1]. It was implemented in Java and successfully tested group sizes up to 1,024 nodes. The success of his project initiated other related work, which addressed some of the limitations of *Hypercube*.

Nicolas Christin and Michael Lack developed *Unicast Hypercube*, a variation on the *Hypercube* protocol, which does not rely on IP Multicast to manage a multicast group. Michael Nahas, developed *DelaunayTriangulation*, a protocol, which relies on Delaunay triangulations to construct and manage a multicast group. Unlike *Hypercube* *DelaunayTriangulation*, adds joining multicast group members in parallel [19]. Christin and Nahas' papers on these protocols are not yet published and therefore the protocols are not described in this paper.

Recent work on this project focused on combining our overlay protocols and constructing an application interface that applications can use. To achieve this, the work was divided in the following three parts. First, additional experiments needed to be performed to verify that *Hypercube* scales well to very large numbers. Beam tested group sizes only up to 1,024 nodes. Before moving on we needed to verify that *Hypercube* scales to even bigger numbers, 10,000 nodes. During the summer of 2000, along with a post-doctorate Dongwen Wang, we successfully reached a *Hypercube* with 10,000 nodes. Data on these large-scale experiments was collected during Fall 2000 and Spring 2001.

Second, *Hypercube* and what later was extended to include the *HyperCast* overlay protocol, design and implementation needed to be more flexible, modular, and scalable. It was redesigned and re-implemented during the summer and fall of 2000 by Liebeherr, Nahas, Wang, and the author. Following good object oriented design techniques we separated the protocol from the implementation detail and still managed to keep a clear relationship between the two [16]. We also incorporated observations from the experiments to make the protocol more scalable.

Third, work began on *HyperCast*, an attempt to merge previous work and provide a mechanism for applications to use it. *HyperCast* provides a socket-like interface to applications through which they can join and send data in a multicast group. Currently, we have an initial implementation of *HyperCast* and several test applications like a multicast File Transfer Protocol (FTP), a game, and a media streamer.

Very recently, Nahas and the author successfully scaled *DelaunayTriangulation* to 10,000 nodes. Preliminary data on these large-scale experiments was collected and is presented in this paper. More extensive testing on the *DelaunayTriangulation* will follow in the near future.

1.5 Preview of the Rest of the Report

Chapter 2 or this technical report, presents the design of *HyperCast* and the design of its overlay protocol structure. Chapter 3 presents the large-scale experiments data on two overlay protocols, *Hypercube* and *DelaunayTriangulation*. Chapter 4 provides the

conclusion, which includes a summary, interpretation for the data collected, and suggestions for future work. Appendix A provides the large-scale experiments data.

Chapter 2: Design of HyperCast

The goal of *HyperCast* is to provide a means for applications, such as e-mail, videoconferencing, distributed whiteboards, to join multicast groups in order to send and receive data in those groups. *HyperCast* abstracts the complexity of multicasting from applications by providing a simple socket like interface. For an application to join a multicast group, it specifies the multicast group address when it creates a *HyperCast* socket. Once it joins a multicast group, it can send and receive data with simple commands. Figure 2-1 shows the abstraction that applications see. Applications A, B and C join the multicast group by opening a *HyperCast* socket. Once they are part of the group, they can send and receive data to each other without knowledge of the underling structure and mechanism.

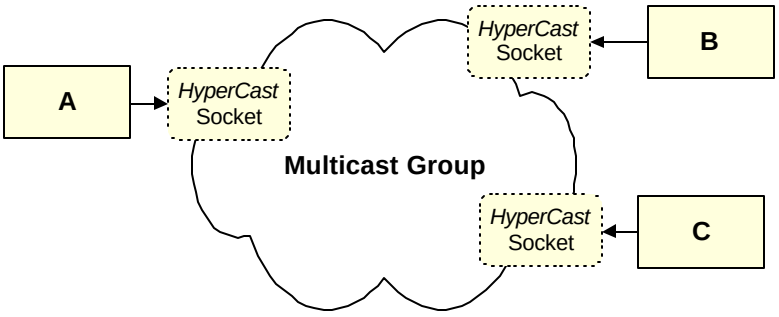


Figure 2-1: Applications' view of *HyperCast*

A *HyperCast* socket is functionally composed of two main parts: an application data-handling part and a multicast overlay part. The application data-handling part is responsible for providing an interface to applications and for sending and receiving

application data. The multicast overlay part is responsible for the exchange of control information between group members in order to build and maintain the multicast group.

2.1 HyperCast Socket

Figure 2-2 shows the components of a *HyperCast* socket. As mentioned earlier, applications communicate with the socket through its interface. The application may manipulate a socket object only through its interface, which allows the application to join and leave a group, send and receive data, look up statistical information, etc. Other components are not visible to the application.

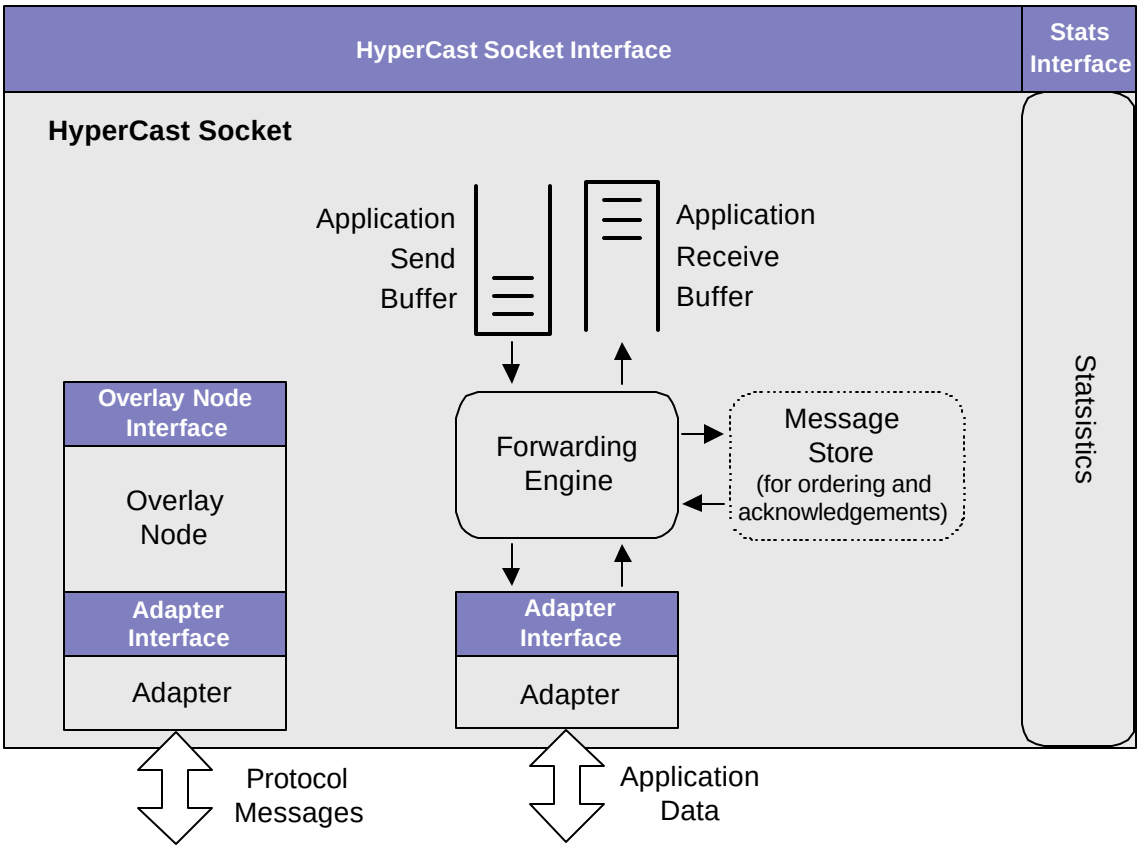


Figure 2-2: *HyperCast* socket components

The application data-handling part consists of the application send buffer, application receive buffer, forwarding engine, message store, and the adapter for application data. The applications send and receive buffers are a place to temporarily store outgoing and incoming data. The forwarding engine, routes incoming and outgoing data between the two buffers, the application data adapter, and the message store. The message store is used for depositing data packets to acknowledge previous packets and to ensure data transmission reliability. The adapter of application data transmits data on the physical network. It knows about the details necessary to actually transmit data. There are various types of adapters that can be used depending on the physical network that the socket is running. For example, if it's running on the Internet then it uses an IP_Adapter (Internet Protocol Adapter). If it is running on a simulator, then it uses a S_Adapter (Simulator Adapter). A more detailed discussion of this topic is provided in the next section. Finally, the statistics object stores statistical information about the socket, like the number of bytes sent, bytes received, etc., which can be queried by the application.

The multicast overlay part consists of the overlay protocol node (OL_Node) and its protocol message Adapter. The overlay protocol builds and maintains the multicast group. This part is essential to *HyperCast*, because the overlay protocol is directly responsible for the scalability of *HyperCast*. Much of the past work has been devoted to the *Hypercube* protocol. The next section provides a detailed design description of the overlay protocol structure. Beside *Hypercube* there are two additional protocols: *Unicast Hypercube*, which unlike *Hypercube* it does not rely on IP Multicasting, and

DelaunayTriangulation, which maps the logical topology closer to the physical topology. The protocol message adapter transmits overlay control data on the physical network. Like the application data adapter, it can be switched with other adapter when running on different physical networks.

2.2 Overlay Protocol Structure

The goal of the overlay protocol is to provide a means for building and maintaining a multicast group. A *HyperCast* socket attaches itself to the overplay topology by creating an overlay node (OL_Node) object in its socket. Once the OL_Node is created, it becomes part of the overlay topology. OL_Node abstracts away the complexity of the overlay topology from the *HyperCast* socket by providing a simple interface to it. For a *HyperCast* socket to join an overlay topology, all it has to do is specify the overlay topology address when it creates an OL_Node object. Once OL_Node joins the overlay topology, *HyperCast* socket can find out the address of its neighbors from its OL_Node to send and receive data from them.

Figure 2-3 shows the abstraction that the *HyperCast* socket sees. A *HyperCast* socket joins the overlay topology by creating an OL_Node, represented by the solid circles, inside the socket. From its OL_Node a *HyperCast* socket finds out about its neighbors. Each *HyperCast* socket has an OL_Node inside it, which correspond with the OL_Node in Figure 2-2. Note that the watermarked part of Figure 2-3 is left there to provide a high level picture of how everything fits.

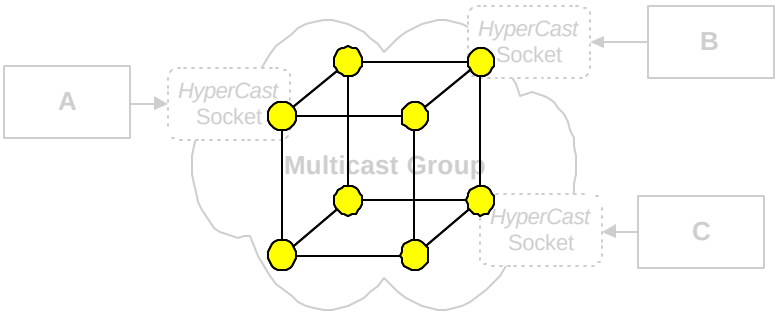


Figure 2-3: *HyperCast* socket's view of the overlay topology

2.3 Overlay Protocol Node

While the original overlay protocol *Hypercube 1.0* successfully tested multicast group sizes up to 1,024 nodes, it had some limitations. These limitations were addressed in the redesign of the overlay protocol and implemented in *Hypercube 1.6*. The new design separates the protocol abstraction from how it is implemented, is very modular and easy to extend other versions from it, and provides a clear relationship between objects through the extensive use of inheritance. The relationships between objects and the modular design is very important since we wanted this design to be easily compatible with future overlay protocols, not just *Hypercube*, and to run on different physical networks, not just the Internet.

From a conceptual level, an overplay protocol can be defined as a set of nodes and the interaction or behavior between those nodes. The overlay protocol is illustrated in Figure 2-4 by the cloud, which contains nodes, represented by the solid circles, and their interaction or the topology they construct, represented by the lines connecting them.

From this model we learn that an overlay protocol needs to have a node object and behavior associated with that node. This philosophy is carried over to our design.

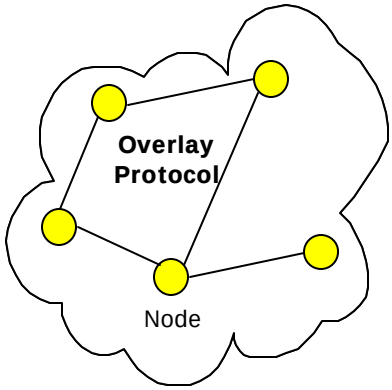


Figure 2-4: Conceptual view of an overlay protocol

All overlay protocol nodes have in common a set of functions or interface. This common interface is abstracted in an interface node (I_Node). In addition, each overlay protocol has a specialized version of I_Node, which encapsulates its specific behavior. For example *Hypercube* has hypercube nodes (HC_Node) and *DelaunayTriangulation* has delaunay triangulation nodes (DT_Node). HC_Node and DT_Node are more specialized versions of I_Node, and are derived from I_Node. HC_Node and DT_Node are said to inherit from I_Node. Figure 2-5 illustrates this relationship. The arrows extending from HC_Node and DT_Node to I_Node indicate that HC_Node and DT_Node inherit from I_Node.

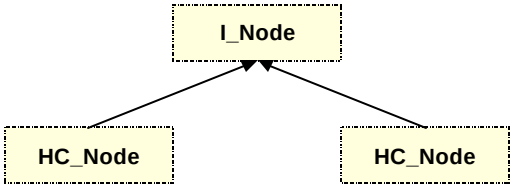


Figure 2-5: Node hierarchy

Up to this point everything is abstract. HC_Node and DT_Node describe the basic notion of the protocol, but do not contain enough information to actually create a node. The information needed to actually create the nodes is dependent on the physical network that it runs on. For example an HC_Node is created differently on an IP network like the Internet than on a simulator network. This is illustrated in Figure 2-6. To create an HC_Node on the Internet we derive a more specialized version of HC_Node, IP_HC_Node (Internet Protocol HC_Node), which contains this implementation detail. Michael Nahas developed a simulator in which we can run and test protocols [18]. If we want to run HC_Node in the simulator, all we have to do is add the simulator detail in S_HC_ Node and inherit from HC_Node.

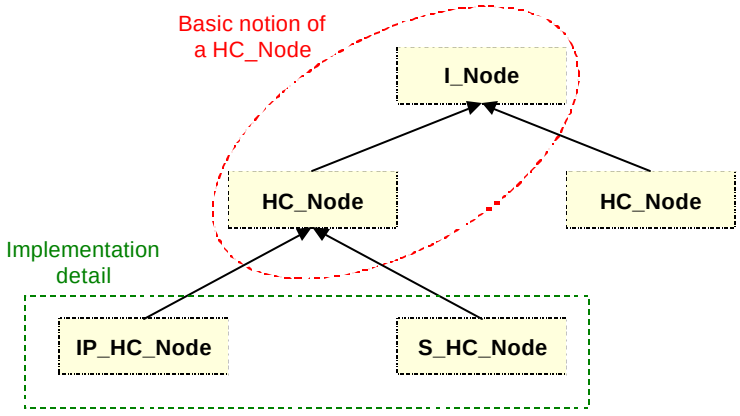


Figure 2-6: Node hierarchy with implementation detail

This implementation detail is contained in an Adapter object. We built a similar hierarchy for Adapters as for Nodes. An I_Adapter specifies the basic interface provided by all adaptors. This includes such functions as getPhysicalAddress(), which all adapters support. Figure 2-7 shows the hierarchy for Internet Protocol (IP) adaptors.

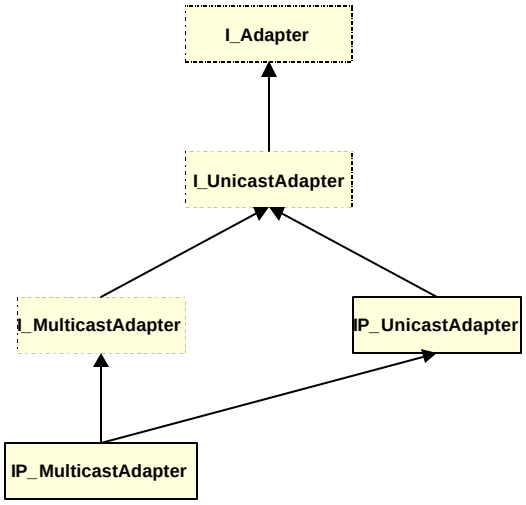


Figure 2-7: Adapter hierarchy

I_Adapter is the most general interface. From I_Adapter we derive an I_UnicastAdapter interface, which only supports unicast transmission. If we are only interested in unicast transmission, then we derive IP_UnicastAdapter from the I_UnicastAdapter interface which contains all implementation details to send and receive unicast data. A multicast adapter, in addition to transmitting unicast data can transmit multicast data. Therefore an IP_MulticastAdapter inherits from both the I_MulticastAdapter interface and the IP_UnicastAdapter.

Each network topology is implemented differently and therefore uses different addressing schemes. Not surprisingly, we created an address hierarchy. In Figure 2-8, I_Address once again is the most general interface. I_PhysicalAddress is for physical topologies, like the Internet and the simulator, and I_LogicalAddress is for logical topologies like *Hypercube* and *DelaunayTriangulation*.

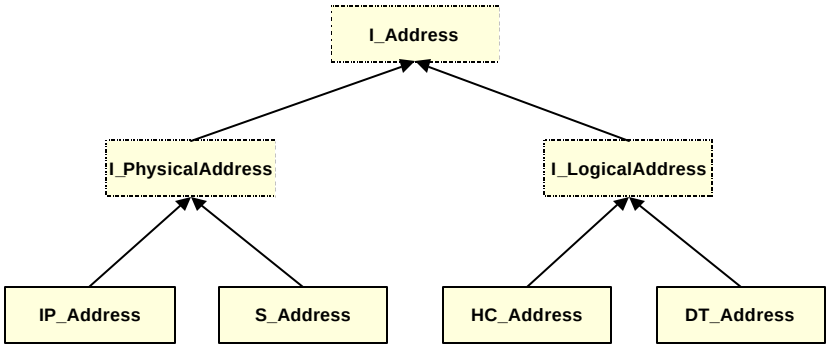


Figure 2-8: Address hierarchy

Messages or packets are exchanged between nodes in the logical topologies. To allow for this we need to define message objects. Figure 2-9 shows the hierarchy for logical topology messages.

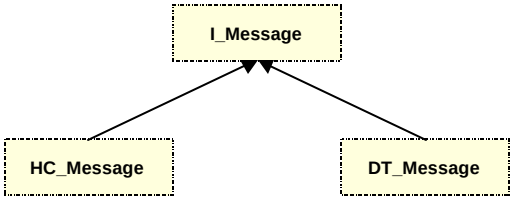


Figure 2-9: Message hierarchy

We now revisit the node hierarchy in Figure 2-10, and show how the relationship between Nodes and Adapters. Besides the I_Node interface, each protocol node needs to have an adapter. Adapters provide the send and receive functions that the protocols use to send and receive data. Although an HC_Node object cannot be created because it is an abstract object, it still needs to have the send and receive functions so that the function calls may be used by the protocol. HC_Node uses IP Multicasting so it has an I_MulticastAdapter. DT_Node does not use IP Multicasting, so it has an I_UnicastAdapter. IP_HC_Node and S_HC_Node can both be created and therefore contain the implementation specific adapters IP_MulticastAdapter and S_MulticastAdapter.

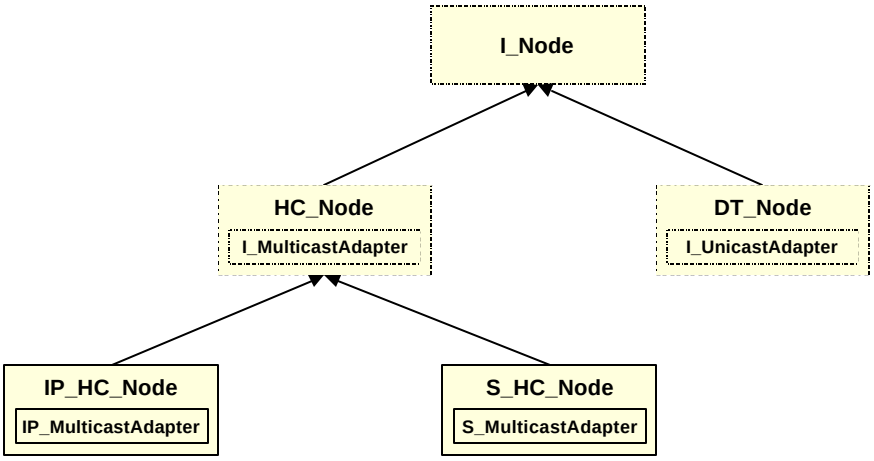


Figure 2-10: Node hierarchy with adapters

Chapter 3: Large Scale Experiments

Hypercube and *DelaunayTriangulation* theoretically scale to thousands and even millions of nodes [4]. Because the theoretical model does not take into consideration many outside factors, such as implementation inefficiencies, implementation errors, and limitations of programming languages, it was uncertain how these protocols would perform in practice and therefore it was necessary to test them.

The scalability properties of the original *Hypercube 1.0* implementation are shown by Tyler Beam in his Master's Thesis, for up to 1,024 nodes [1]. His results closely match the theoretical predictions. While in our new implementation, *Hypercube 1.6*, the basic protocol did not change, the underlying implementation structure has been fundamentally modified. Scalable software is very sensitive, where even minor changes can have significant implications. Therefore, it was necessary to verify that our new design did not introduce any problems. In addition we wanted to increase the number of nodes by an order of magnitude to 10,000 nodes. With *Hypercube 1.6* scaling to 10,000 nodes, we are more confident that it will scale to hundreds of thousands or even millions of nodes.

Unlike *Hypercube*, *DelaunayTriangulation* was not previously tested on the Internet. Michael Nahas tested the protocol in his simulator with up to 512 nodes [19]. Given the fact that these tests were not performed on a real network and that the number of nodes was relatively small, it was even more necessary than with *Hypercube* to verify that *DelaunayTriangulation* scales on a real network.

3.1 Testbed

The testbed for the experiments is Centurion, a large computer cluster at the University of Virginia [6]. The part of the cluster used for the experiments is composed of 110 machines running the Linux operating system. These machines are Dual 400 MHz Pentium II with 128 MB of RAM and are connected with a 100Mbits/sec Ethernet switched network. In addition to the machines mentioned above, there are several other machines called *frontends*. Users log on to one of the Centurion *frontends*, to submit jobs to the rest of the cluster.

3.2 Performing an Experiment

In addition to the protocol (*Hypercube* or *DelaunayTriangulation*) nodes, two programs are involved in running an experiment. The first program, Runcontrol, starts the experiment, monitors the experiment, collects data and ends the experiment. The second program, Runserver, creates protocol nodes on the physical machines and acts as an intermediary between Runcontrol, and the nodes it created. For *DelaunayTriangulation* each Runserver creates up to 100 nodes with 1 Runserver started on each machine for a total of 100 nodes. For *Hypercube* each Runserver creates 32 nodes and up to 3 Runservers are started on each machine, for a total of 96 nodes per machine. For *Hypercube* to run these many nodes on a machine, we had to optimize the protocol implemented in Java using the TowerJ static compiler [7]. The TowerJ static compiler takes Java bytecode and converts it to optimized native machine code. The conversion allowed us to run two to three times more node per machine [10].

Figure 3-1 shows the setup for an experiment. Runcontrol, represented by the desktop computer, runs on one of the Centurion *frontend* machines. Runservers, represented by the tower computers, are started on the rest of the machines. Each Runserver creates protocol nodes, which are indicated by the small circles under the Runservers. Information is exchanged between Runcontrol and Runserver during the experiment, which is depicted by the double-pointing arrows.

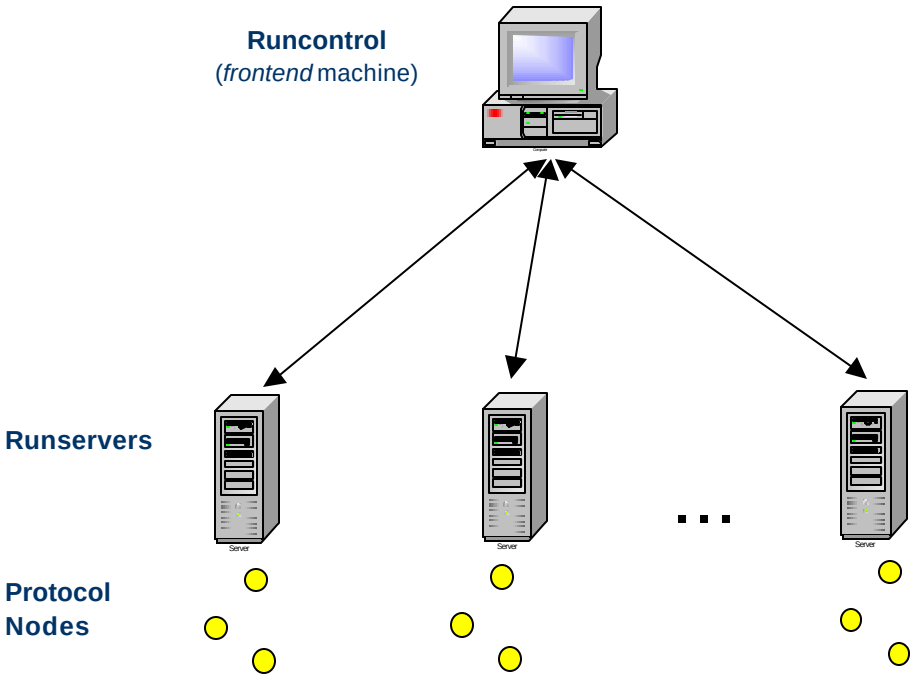


Figure 3-1: The schematic representation of an experiment.

The first step in performing an experiment is to log on to Centurion through one of the *frontend* machines. Second, the availability and load of the rest of the Centurion machines is checked. Machines that are down or that are used heavily by other people cannot be used for the experiment. Third, the *frontend* machine starts *Runcontrol* and

commands the rest of the machines to remotely start Runservers. Fourth, it is necessary to carefully monitor the experiment while it is running to make sure that nothing goes wrong. Some of these experiments can take up to 20 hours to complete, which presents a challenge for the monitor. If something does go wrong with the experiment, it can take down all machines on which it is running. Users from all over the world share these machines and the failure of a significant portion of Centurion will affect most current users. Finally, once the protocol reaches a stable state, data is collected and the experiment ends.

3.3 Types of Experiments

For the large-scale experiments, we were interested to verify that the protocols scale, and the time it takes them to grow from 0 to N nodes, where N is large. For *Hypercube* we also analyzed the effect of new nodes joining an already stable *Hypercube*.

3.3.1 Time to Stabilize Hypercube

This experiment tests whether or not the protocol scales to large numbers, i.e. with group membership of 10,000 nodes. It also shows the rate at which the *Hypercube* grows as a function of its size. In addition it illustrates the effect of adding a large number of joining nodes to an existing *Hypercube*, and whether or not such a perturbation can cause the *Hypercube* to collapse.

3.3.1.1 Description of the Experiment

In part one of the experiment, we started with an initially empty *Hypercube* and measured the time it takes the protocol to stabilize with N number of nodes. N was increased in powers of two from 32 nodes to 10,000 nodes. The protocol is considered stable when all N nodes have successfully joined the *Hypercube* and are in the stable state.

In part two of the experiment, we started with M number of nodes already stable in the *Hypercube*, and N the number of joining nodes. We are interested in large-scale experiments so M was varied in powers of 2 from $2^{10} = 1024$ nodes to $2^{14} = 8196$ nodes. The number of joining nodes N, was independently varied in powers of 2 from $2^{10} = 1024$ nodes to $2^{14} = 8196$ nodes. Because our testbed could support only a little over 10,000 nodes, M and N were varied such that $M + N \leq 10{,}240 = 2^{11} + 2^{13}$ nodes.

3.3.1.2 Data and Interpretation

The results for part one of the experiment are graphed in Figure 3-2. The relationship between the number of nodes (x-axis) and the time to stabilize the *Hypercube* (y-axis) is linear. This is expected since nodes are added to the *Hypercube* one at a time.

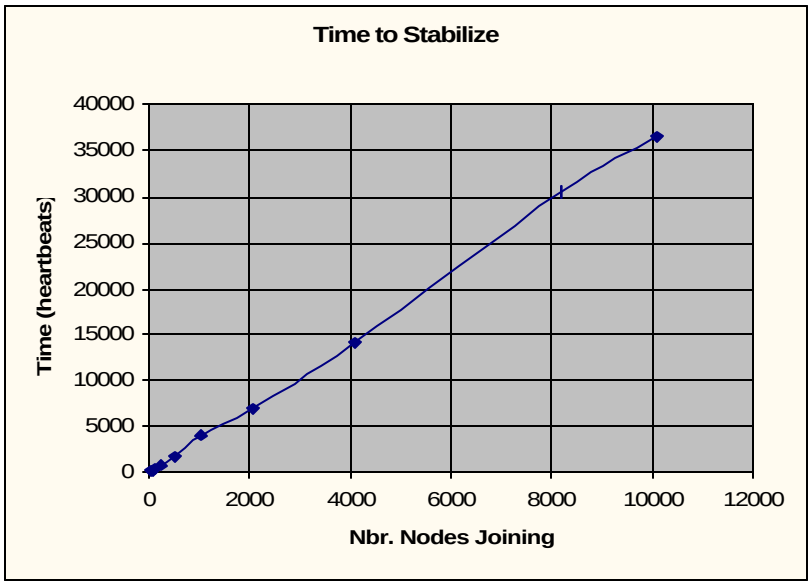


Figure 3-2: Time to stabilize a *Hypercube* with N nodes

The average time to add a node to the *Hypercube* is shown in the Figure 3-3. The graph is logarithmic to clearly show the average time to add a node for the wide range of the number of nodes joining. As it is apparent from the figure, this number fluctuates slightly around 3.5 heartbeats. This means that every 3.5 heartbeats a node is added to the *Hypercube* and reaches a stable state. The 3.5 heartbeats comes from the state diagram of *Hypercube*. A new node goes through 3 states before it is in a stable state: joining, incomplete, and stable [1]. The rest of the 0.5 heartbeats comes from such things as overhead, multiple nodes try to join at the same time, and packets being lost.

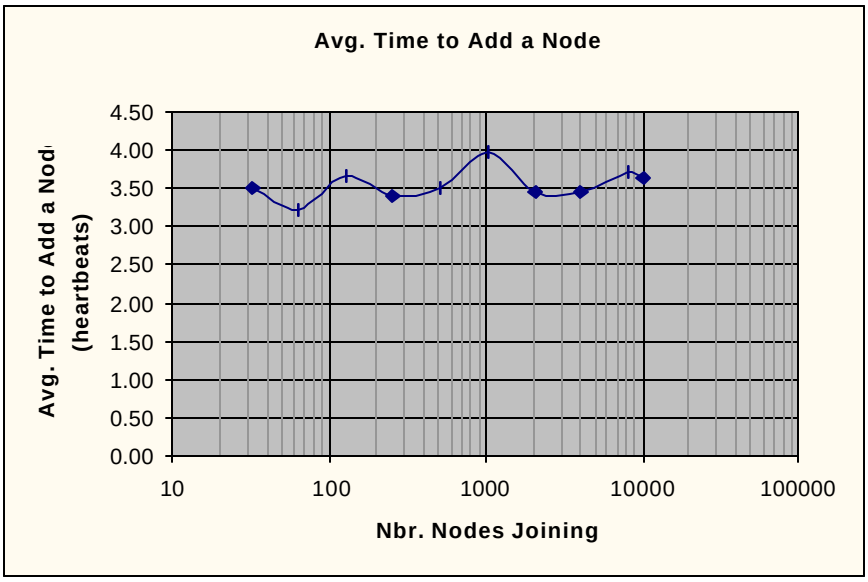


Figure 3-3: Average time to add a node to the *Hypercube*

The results for part two of the experiment are graphed in Figure 3-4. The number of stable nodes and the number of joining nodes are independently varied. As it is apparent from the graph, the time to stabilize is independent of the number of nodes already stable in the *Hypercube*. This is expected since as Figure 3-3 shows, the average time to add a node to the *Hypercube* is independent of the size of the *Hypercube*. The exponential curve on the Nbr. Joining Nodes axis, depicts a linear growth since it is a log scale and therefore agrees with the linear relationship is Figure 3-2.

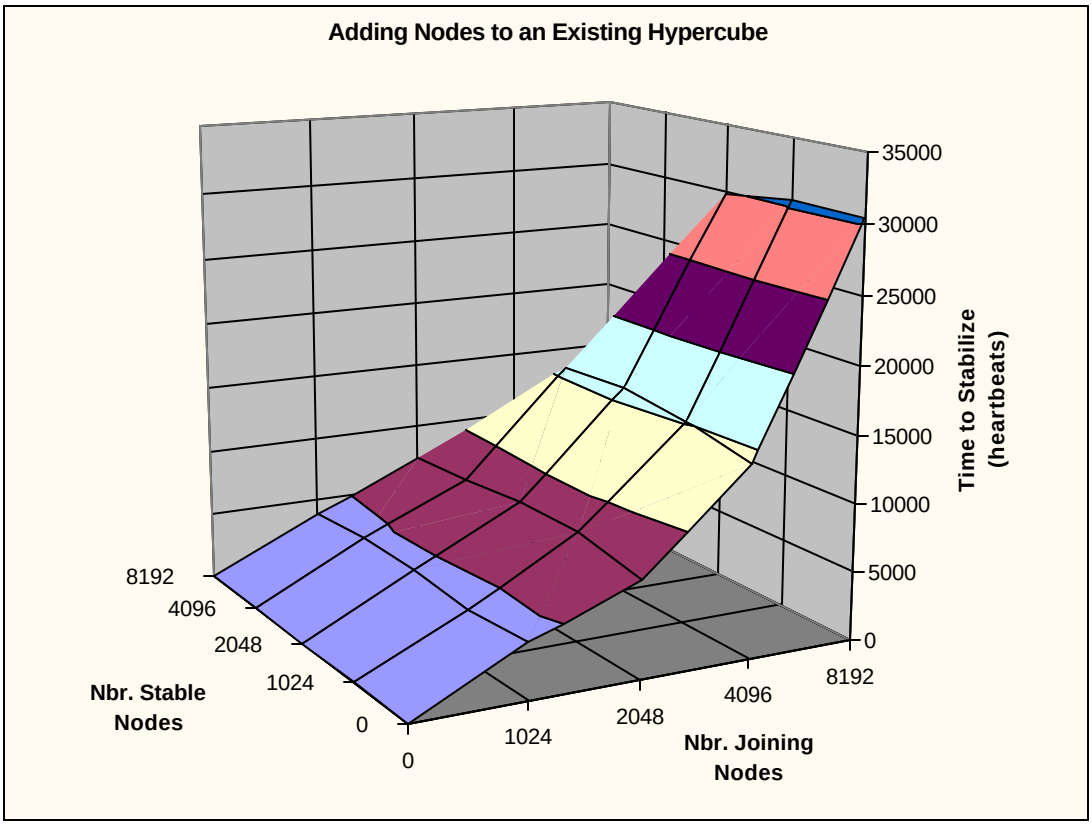


Figure 3-4: Adding nodes to an existing *Hypercube*

3.3.2 Time to Stabilize DelaunayTriangulation

This experiments is testing weather or not the protocol scales to large numbers, i.e. with group membership of 10,000 nodes. It also shows the rate at which *DelaunayTriangulation* grows as a function of its size. In addition it illustrates the average traffic per node per heartbeat while the *DelaunayTriangulation* is being constructed.

3.3.2.1 Description of the Experiment

We start with an initially empty *DelaunayTriangulation* and measure the time it takes the protocol to stabilize with N number of nodes. N is doubled each time from 312 nodes all the way up to 10,000 nodes. The protocol is considered stable when all N nodes have successfully joined the *DelaunayTriangulation* and are in the stable state.

3.3.2.2 Data and Interpretation

The results for part one of the experiment are graphed in Figure 3-5. The relationship between the number of nodes (x-axis) and the time to stabilize the *DelaunayTriangulation* (y-axis) is not very clear, but appears to be partially linear. The protocol seems to grow at an inverse square rate up to about 5,000 nodes. However, this rate seems to become linear around 10,000 nodes. It is difficult to tell exactly what the relationship is from these preliminary results because there is insufficient data at this time. Additional future tests with a wider variety of data points should clearly reveal the time to stabilize relationship.

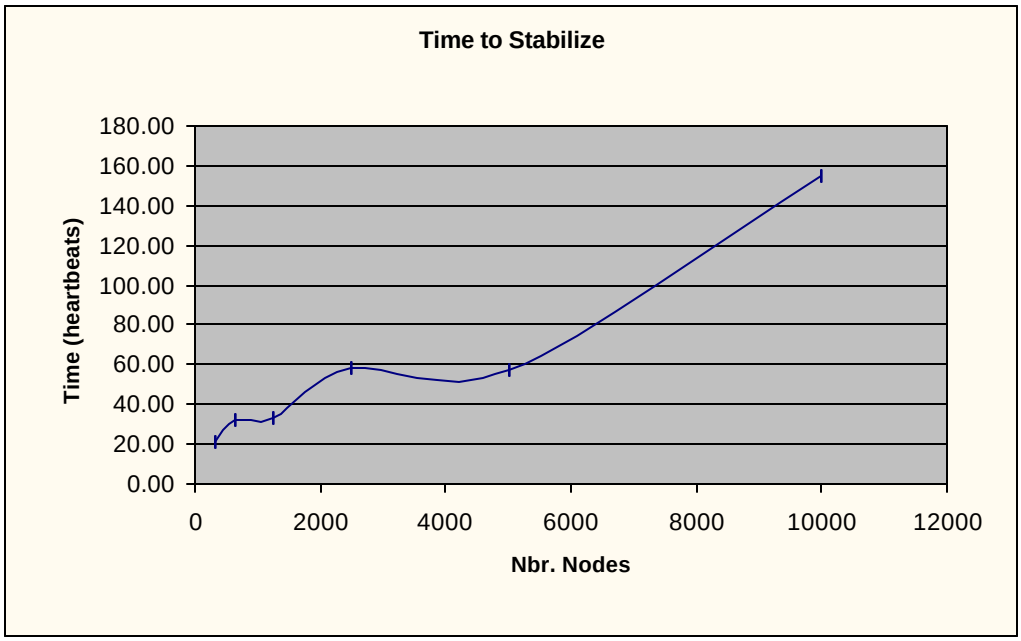


Figure 3-5: Time to stabilize a *DelaunayTriangulation* with N nodes

This growth trend is better illustrated in Figure 3-6, which plots the average time to add a node as a function of the *DelaunayTriangulation* size. The time to add a node to the *DelaunayTriangulation* seems to decrease exponentially up to about 5,000 nodes. This agrees with the fact that nodes are being added in parallel. However, after about 5,000 nodes, the average time to add a node seems to level of between 0.01 and 0.02 heartbeats.

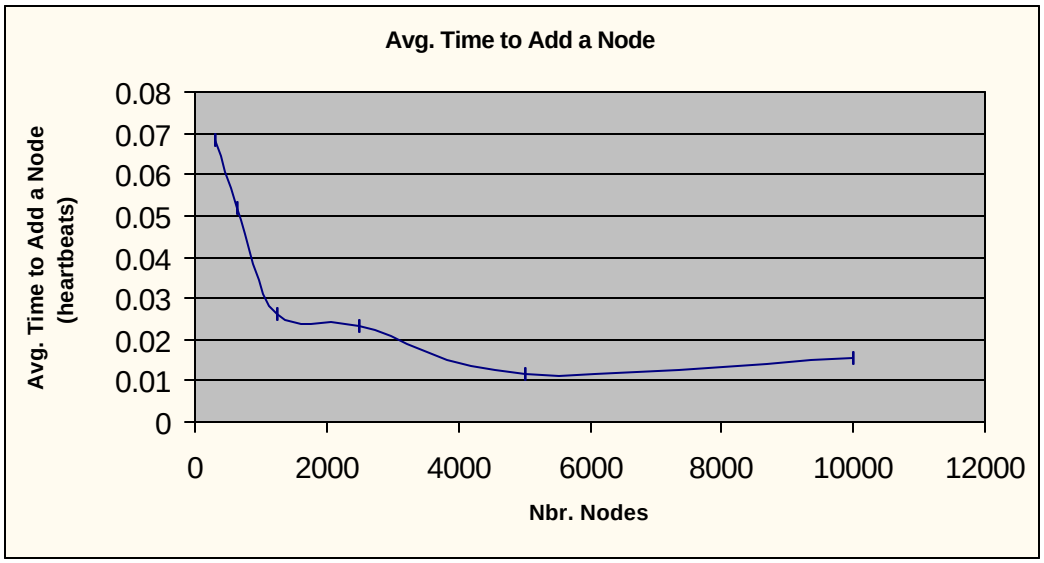


Figure 3-6: Average time to add a node to the *DelaunayTriangulation*

The number of bytes sent and received at each node per heartbeat is graphed in Figure 3-7. The fact that the amount of traffic sent and received at each node is a constant as a function of the number of nodes present, shows that the protocol scales well. This is expected since each node has an average of less than 6 neighbors, for any given group size [19].

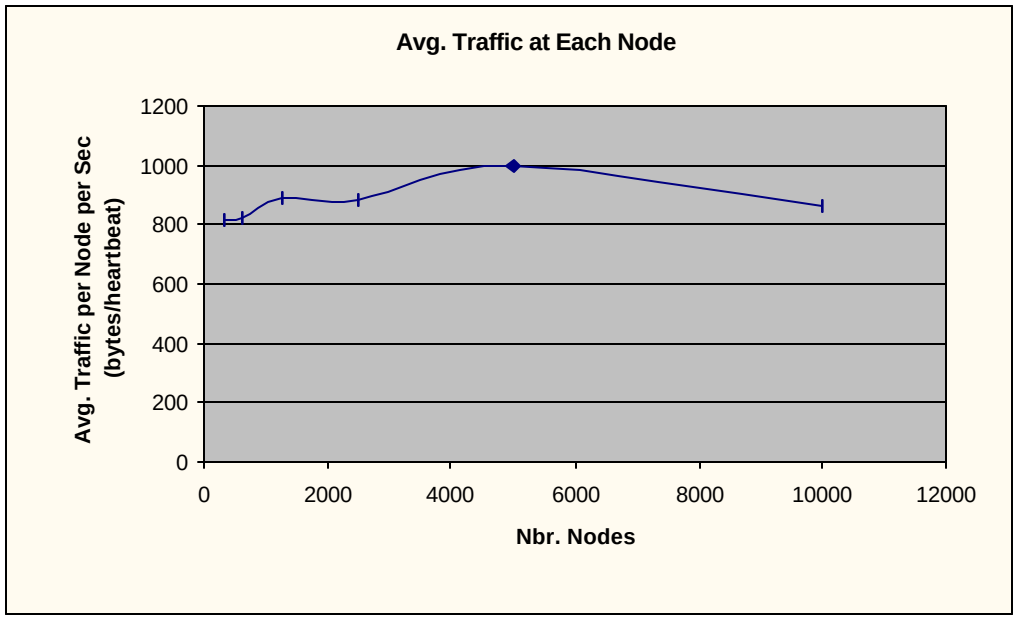


Figure 3-7: Average bytes sent and received per node per second

Chapter 4: Conclusion

Existing scalable multicast protocols do not efficiently handle multicast groups with many senders. Protocols that do address situations with many senders do not scale above a few hundred members. *HyperCast* fills this gap by allowing for very large multicast groups with many senders.

4.1 Summary

This paper presented the design of *HyperCast* from a very high level as seen by applications to the level of individual objects that make up *HyperCast*. *HyperCast* provides a socket-like interface abstraction to applications, which hides the complexity of the protocol. *HyperCast* socket is functionally composed of two parts: an application data handling part and an overlay part. The overlay part is responsible for building and maintaining the multicast group. OL_Node provides an interface abstraction to *HyperCast* socket, which hides the complexity of the overlay protocol. OL_Node can run different overlay protocols like *Hypercube* or *DelaunayTriangulation*. Next, the design of the overlay protocol or OL_Node is presented. This design provides a clear relationship between the different objects making up an overlay protocol through the extensive use of inheritance. This design framework provides a very easy method for extending or modifying existing protocols and the introduction of new protocols. Some of these are changing the Adapter to run on different physical networks, running unicast or IP Multicast versions and switching the protocol node to run a different overlay protocol.

Large-scale experiments were performed on two overlay protocols *Hypercube* and *DelaunayTriangulation.* The *Hypercube* experiments verify that the protocol scales to 10,000 nodes. The time required by the protocol to reach 10,000 nodes was 35,000 heartbeats or 19.4 hours with the heartbeat set to 2 seconds. The *DelaunayTriangulation* experiments in turn show that it also scales to 10,000 nodes. The time required by the protocol to stabilize is 155 heartbeats or 5.2 minutes with the heartbeat set to 2 seconds.

4.2 Interpretation

The socket-like interface provided by *HyperCast* to applications makes it very easy to use. In fact from an application's perspective it is no different than using any other socket. The flexibility of the overlay protocol structure, makes *HyperCast* a "dynamic" protocol that can be very easily extended to fit individual needs. As with Object Oriented languages like C++ and Java, it is the ability to extend these languages through the creation of new objects types that makes them so powerful and successful. *HyperCast* follows the same principle by giving the users great flexibility and the ability to very easily extend it.

In our *Hypercube* experiments, we successfully tested group members with 10,000 nodes. Although this is an order of magnitude increase from Beam's 1,024 node experiments, the data trends remained the same. For example, the rate at which nodes are added to the hypercube is the same for 10, 1000, and 10000 nodes, about 3.5 heartbeats per node. In addition Figure 3-4 shows that the perturbation of adding a large number of joining nodes

to a stable hypercube is very small and does not cause it to collapse. These continuing trends for large experiments provide a high degree of confidence that the protocol scales to even larger numbers. One of the advantages of *Hypercube* is that each node is guaranteed to have $\log_2 N$ neighbors, which ensures that the complexity at each node grows logarithmically. This also implies that the maximum distance from any node to any other node is $\log_2 N$ hops away. The drawback to this protocol is that nodes are added one at a time and therefore take a long time to stabilize a large hypercube. This makes it also very hard to test and debug, because the turnaround time between making modification to the code and testing it can take over 15 hours. Although *Hypercube* takes relatively a long time to stabilize, it provides some very strong guarantees once it is stable and therefore it can be used in applications, which stay up for a long time. Such applications might include distributed search engines and large parallel computers.

Although *DelaunayTriangulation* was previously untested on the Internet, we were able to scale it up to 10,000 nodes in a surprisingly very short amount of time. It only took several weeks as opposed to several months with *Hypercube*. Part of the success was because we already had all necessary resources available and running, gained experience from running *Hypercube* experiments, and had a short turnaround time between making modifications to the code and testing it. One of the main advantages of *DelaunayTriangulation* is that joining nodes can be added in parallel and therefore stabilizes very fast. However, unlike *Hypercube, DelaunayTriangulation* does not guarantee a maximum number of neighbors for each node nor a maximum number of hops from one node to another. On average and in most cases, nodes have less than 6

neighbors but in some cases this number can be very high [19]. Because of the fast time to stabilize, *DelaunayTriangulation* can be used by applications that require fast construction of multicast groups. Usually these applications involve a lot of user interaction such as videoconferencing or whiteboards.

4.3 Recommendations and Future Work

The very recent success of scaling *DelaunayTriangulation* to 10,000 nodes did not provide enough time to thoroughly test it. Preliminary data was collected and presented in this paper, but a lot more experiments remain to be performed. Some interesting experiments are to measure the average and maximum burst rates at which nodes contact the web server as a function of the group size, observe how the protocol behaves if a large number of nodes fail, and the effect of varying the heartbeat. My hypothesis is that *DelaunayTriangulation* can reach group sizes well over 10,000 nodes on Centurion, since there were still resources available when we run *DelaunayTriangulation* with 10,000 nodes.

The next step in large-scale experiments is to test these protocols over wide area networks. Unlike in local area networks, computers in wide area networks can be very far from one another and therefore the delay between machines is a lot higher. This can have interesting effects on the protocols.

The data store component of *HyperCast* socket is used to provide reliability by merging ACK or NACK packets before sending them up the message tree. For very small groups,

a node can wait to hear back from all its children before merging and sending ACKS or NACKS up the tree. However, for very large groups, some children may be a lot slower than others at responding. In fact if nodes fail, they may not even get an ACK or NACK. How long a node should wait before it merges and sends ACK or NACK replies, is a difficult question to answer. There is no perfect solution or policy to this problem since the unreliability tends to increase with the size of the group. What constitutes a good solution changes with group size. Because of this constant change an approach worth exploring is to devise a general policy. One solution might be to change the policy automatically as the group size changes. Another might be to have the sender specify the policy in the message packet sent, and have the nodes return ACKS or NACKS based on that policy.

Hypercube and *DelaunayTriangulation* have their advantages and disadvantages. An interesting research problem is to combine the two protocols in such a way as to explore their advantages. Use *DelaunayTriangulation* to quickly reach a stable state, and once it is stable run a mapping algorithm, which converts *DelaunayTriangulation* to *Hypercube.* This would take advantage of *DelaunayTriangulation's* fast stabilization time and *Hypercube's* steady state guarantees.

References

[1] Beam, Tyler K. "HyperCast: A Protocol for Maintaining a Logical Hypercube-Based Network Topology." Master's Thesis. University of Virginia. 1999.

[2] Francis, Paul. "Yallcast: Extending the Internet Multicast Architecture." NIT Information Sharing Platform Laboratories. September 1999. Available: http://www.yallcast.com/.

[3] Liebeherr, Jörg., and Beam, Tyler K. "HyperCast: A Protocol for Maintaining Multicast Group Members in a Logical Hypercube Topology." Proceedings of the First International Workshop on Networked Group Communication (NGC '99), in: Lecture Notes in Computer Science, Vol. 1736, pp. 72-89, July, 1999.

[4] Liebeherr, Jörg., and Sethi, B. S. "Towards Super-Scalable Multicast." Technical Report. CATT 98-121. Polytechnic University. January 1998.

[5] Mauve, Martin., and Hilt, Volker. "An Application Developer's Perspective on Reliable Multicast for Distributed Interactive Media." University of Mannheim, 1999.

[6] Centurion computer cluster. "Centurion" March 2001. Available: http://legion.virginia.edu/centurion/

[7] TowerJ corporation. "TowerJ" March 2001. Available: http://www.towerj.com/.

[8] Levine, B. "A Comparison of Known Super-Scalable Multicast Protocols". Masters Thesis. University of California Santa Cruz, June 1996.

[9] Tanenbaum, Andrew. Computer Networks. 3rd ed. Prentice Hall. Upper Saddle River, NJ, 1996.

[10] Lorincz, Konrad. "TowerJ vs. JVM Performance Report". University of Virginia. March 2001. Available: http://www.cs.virginia.edu/~kel9m/Research/HyperCast/TowerJPerformanceReport/.

[11] R. Yavatkar, J. Griffioen and M. Sudan. "A Reliable Dissemination Protocol for Interactive Collaborative Applications". Proceedings of ACM Multimedia '95, November 1995.

[12] Quinn, Michael J., Parallel Computing Theory and Practice. New York, NY: McGraw Hill, 1994.

[14] Peterson, Larry L., and Davie, Bruce S. Computer Networks A Systems Approach. San Francisco, CA: Morgan Kaufman Publishers, 2000.

[16] Strustrup, Bjarne. The C++ Programming Language. 3rd ed. Reading, MS: Addison Wesley, 1997.

[17] Liebeherr, Jörg., and Sethi, B. S. "A Scalable Control Topology for Multicast Communications". Proceedings of IEEE Infocom 1998.

[18] LoToS network simulator March 2001. Available: http://www.cs.virginia.edu/~jorg/research/hypercast/.

[19] Liebeherr, Jörg., and Nahas, Michael. "Application-layer Multicast with Delaunay Triangulations". Submitted to IEEE Globecom 2001. March 2001.

Appendix A – Experiments Data

In electronic form.