Integrating FreePastry and HyperCast

Greg Mattes
Spring 2004
Independent Networks Study
Overlay Protocol Implementation in HyperCast
Prof. Jorg Liebeherr
University of Virginia

Introduction

This document lists the specific issues involved in integrating the FreePastry code base into the HyperCast system. For a general overview of integrating arbitrary overlay protocols into HyperCast see REF.

Implementing I_Node APIs

In FreePastry routing is performed when a "route message" is processed in the system. There is no API to ask for the next hop to a given node. In order to implement the getParent() API of I-Node interface, the code for the Pastry routing algorithm had to be removed from the file StandardRouter.java and placed in HyperCastNodeAdapter.java with minor modifications.

All neighbors is defined to be everybody in the leaf and route sets. There is support in HyperCastNodeAdapter.java to merge these two sets and return their union.

Timer Adapter

Pastry uses java.util.Timer (Java 1.3), whereas HyperCast implements its own Timer class (in the adapter directory). Both do essentially the same thing, but there are at least three major differences. First, the HyperCast Timer class gives no API that allows the programmer to specify that a certain event should continue to occur periodically, rather the burden is on the programmer, when handling a timer event, to reschedule that event. Pastry calls the java.util.Timer APIs "scheduleAtFixedRate" and "schedule." These two APIs do slightly different things, one is fixeddelay and the other is fixed-rate (See the Javadoc for java.util.Timer). The HyperCastTimer provides a timerExpired method for use with previously scheduled HyperCastScheduledMessages. The timerExpired method should be called on all HyperCastScheduledMessages to ensure that they are rescheduled properly if they were defined by Pastry to be periodic. The HyperCastTimer class does not make the distinction between fixed-rate and fixed-delay scheduling, however, it is implemented as fixed-rate (to make it fixed-delay simply move the line of code "msg.runO;" to be the first line of code in the timerExpired method. If the difference between fixed-rate and fixeddelay is crucial in the future, two APIs can be easily defined with the "msg.runO" as the first line of one API and the last line of the other.

Second, the HyperCastTimer does not run in its own thread as does the Pastry timer (since Pastry uses java.util.Timer). One of the reasons for having the HyperCastTimer class is so that Pastry timer events happen in the HyperCast adapter's timer thread.

Finally, the Java Timer class works with TimerTask objects rather than general Objects as is the case in HyperCast. TimerTask objects have associated with them a run method (this does not imply that TimerTask is derived from Runnable, i.e.. a TimerTask does not have its own thread, rather it uses the java.util.Timer thread). This run method must be invoked in order for the TimerTask registered with the Timer to do its work. In the HyperCast case, the programmer writes a handler that does whatever is appropriate with the Object that is passed to the timerExpired method of the I-AdapterCallback interface, i.e.. the code run by a HyperCast timerlD object is not usually stored with the object, but rather with a handler, a TimerTask has a built in handler called "run." The HyperCastTimer method timerExpired takes care of invoking the run method of a Pastry TimerTask (Pastry extends "TimerTask" to be a "ScheduledMessage," there is another class called "HyperCastScheduledMessage" that is extended from "ScheduledMessage" which is used in this interface.

Order of Initialization Operations/Join

In HyperCast when a node is created its data structures are created first, then a rendezvous mechanism finds another node already in the overlay, finally, the overlay join operation proceeds.

This is not the way that FreePastry is architected, FreePastry transposes the order of the first two operations, rendezvousing with an existing node first, then initializing data structures. This implies that in FreePastry rendezvous messages do not pass through the node network interface as they do in HyperCast, specifically, through the node adapter.

In addition to rendezvous, Pastry exchanges a series of messages that are not part of the Pastry overlay protocol in order to locate an existing node that is physically close to the the new node. Only after such an existing nearby node is identified can the data structures for a new pastry node be created because in the architecture of the FreePastry system the initialization of these data structures requires a nearby node to be known. I refer to this as the first phase of join.

Only after the first phase of join has completed does the Pastry protocol join operation proceed. I call this the second phase of join.

The second phase of join are simply Pastry overlay protocol message so they are handled easily by passing them to the FreePastry system for processing. The first phase of join message however are very difficult to implement using HyperCast. Unfortunately no code from FreePastry can be reused for this phase of join because FreePastry uses bi-directional communication channels that HyperCast does not support. The code for the first phase of join is implemented as a state machine in `PastryJlode-BuddyList. java.`

Pastry H HyperCast Node Adapter

This is not the node adapter in HyperCast that is responsible for overlay protocol messages, rather this is the class that is used as a bridge between the HyperCast `Pastry-Node` class and the Pastry `HCPastryNode` class. It allows messages to be passed between the code of the two systems. Another function provided by this adapter is address translation back and forth between `I-PhysicalAddress` used by HyperCast and `InetSocketAddress` used by Pastry.

Logical Address

A class was created to support Pastry logical addresses in the file `Pastry_LogicalAddress. j ava.` This class implements the adapter pattern and forwards most method calls to the FreePastry `Node Id class.`

Code in `Id. java` had to be changed so that the number of bits used in a Pastry logical address was 128 instead of 160. This is due to a logical address length constraint imposed by the HyperCast message format.

Rendezvous

A "buddy list" rendezvous mechanism was chosen to be used with Pastry in HyperCast. This mechanism is simple to use and to implement, indeed an existing buddy list implementation from the HyperCast Delaunay triangulation code is used as the basis for the implementation used with Pastry.

With buddy lists, a set of well known buddies is placed in a static configuration file that is read on system initialization. Attempts are made to contact these buddies, one-by-one, until contact is established. When contact is established the two phases of Pastry join begin.

Unlike the Delaunay triangulation code, the Pastry buddy list code supports only "static" buddies and does not cache buddies in a file.

Message Formats

The class `Pastry-Message` was implemented to be able to pass Pastry messages among HyperCast nodes. Since FreePastry is used as a "black-box," Pastry overlay protocol messages are not interpreted by the HyperCast code. The `Pastry-Message` class is responsible for formatting and reading HyperCast message headers, however the content of Pastry overlay message is handled by
FreePastry code.

Java serialization is used to create byte streams that represent Pastry overlay messages. These byte streams are passed to HyperCast when Pastry needs to send information. The byte streams are included as an opaque payload of a `Pastry-Message.`

In addition to overlay messages, several message types used in rendezvous the first phase of join

are defined by `Pastry-Message.` These messages were originally implemented in `PastyNodeFactory. j ava`
but as explained in the Join section of this document, could not be used.

Multicast

Multicast was not implemented using Pastry in HyperCast. The operations for determining which neighbors are children of an arbitrarily based multicast tree cannot be done locally on a Pastry node. Messages to all potential children, i.e.. all neighbors, must sent and those potential children must report back to the sending node if that node is not their parent. In this way a tree may be built.

HyperCast Improvements

Various changes were made to the HyperCast code base as a result of this work. The Pastry API specifies that node should be notified when is has complete a join operation. HyperCast had no support for such a mechanism so one was added.

An operation called "previous hop check" was added to HyperCast to aid in the build of a multicast tree in protocols, like Pastry, that cannot make a local determination of which neighbors are children in an arbitrary tree.

A "factory" pattern was added to HyperCast for use in the creation of nodes. It was noticed in the analysis of the FreePastry code that the factory pattern was useful when creating nodes with different network drivers. This concept was transferred to HyperCast when creating a node that is one of many types.