Chimera: A Library for Structured Peer-to-peer Application Development

Matthew S. Allen
msa@cs.ucsb.edu

Rama Alebouyeh
rama@cs.ucsb.edu

1 Introduction

In this paper, we describe the present state of our work in implementing Chimera, a library that implements a structured, peer-to-peer system. This work is an attempt to provide a library that allows easy development of applications on top of a peer-to-peer routing infrastructure. The goals are twofold. First, we wanted to make a fast, lightweight, C implantation of a Tapestry-like system includes some of the optimizations provided by other systems. Second, we wanted to develop a system designed to export an API in line with existing work that describes how to effectively interface with such an overlay network.

Chimera succeeds at these goals. The library implements a routing infrastructure much like those provided by Tapestry and Pastry. The system contains both a leaf set of neighbor nodes, which provides fault tolerance and a probabilistic invariant of constant routing progress. It also provides a PRR style routing table to improve routing time to a logarithmic factor of network size. Using this library, developers can build an application that creates an overlay network with a limited number of library calls. They can implement their own application by providing a series of upcalls that are called by the library in response to certain overlay network events.

The library we developed will serve as both a useable interface and a starting point for further research. This library implements a relatively complete version of a structured peer-to-peer system we described. It includes some of the current work in locality optimization and soft-state operations. It also provides an interface that can be used as is to develop applications, but that will allow for the infrastructure to be changed with little impact on existing application code.

This paper will proceed as follows. In section 2, we will describe the components of our system and the design decisions that led to the current state of the implementation. In section 3, we will describe the specific API that the current system exports. And, in section 4, we discuss how this system currently performs.

2 System Design

The Chimera system is divided up into a collection of components that can be modified independently with little impact on the system. *Figure 1* shows these components and the points of communication between them. In this section, we will describe the various components and how they interact with each other. We will also discuss the rational between current design decisions and discuss some improvements that could be made to the system.

2.1 Routing System

The core of the Chimera library is the routing system, which implements a structured peer-to-peer overlay similar to Tapestry or Pastry [5, 4]. These algorithms are described thoroughly in other papers, so we include only a brief overview of them.

A peer's routing information consists of two data structures: a leaf set and a PRR routing table [2]. The leaf set contains the current peer's immediate neighbors in key space. This leaf set is augmented by a PRR style prefix-matching routing table. When a message arrives that needs to be routed, the local node checks its leaf set first. If the destination falls in the leaf set range, it will be forwarded to the closest node in the leaf set. Otherwise, it will be routed through PRR routing table to the node with the longest common prefix with the destination.

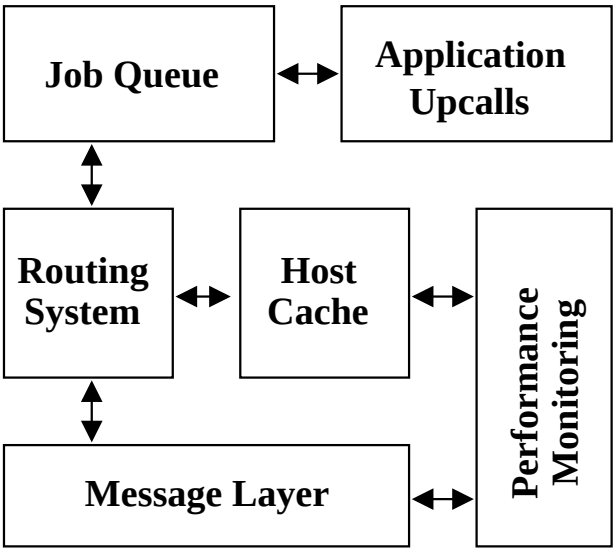

Figure 1: Chimera system design

Peers join the system by contacting a *bootstrap* node. After being assigned a key, the node's join message is routed through the network to the peer that will be closest in key space to the joining node. The joining peer then inherits its initial routing information from this peer. Through the peer's lifetime, it augments its routing information with peers that it interacts with or learns about. Leaf sets are strictly constructed, and must contain the closest known peers in ID space for the system to work. PRR Routing tables have less stringent requirements, and Chimera exploits this by updating this table with nodes with the best expected performance. Performance numbers are acquired through the *performance monitoring* subsystem.

This system is implemented using the soft-state mechanisms used in the Bamboo system [3]. This means that newly joined hosts are not added to anyone routing table or leaf set until they have a complete leaf set. Information about new hosts propagates relatively slowly, so hosts are not likely to be found in routing tables until they have been in the system a long time. Finally, operations performed by the Chimera system are idempotent and non-atomic, so mid-operation failures can be handled by redundant messages without the risk of putting the system in an unstable state.

The routing system is relatively complete, but there are many features implemented by other systems that are lacking. At the present time, keys for hosts and messages are only 32 bits as opposed to the 160 bits most systems support. Also, failure detection and recovery are fairly weak and could stand to be improved. Specifically, the system could easily support multiple alternate hosts for each PRR routing table entry, but currently doesn't. There are many other improvements based on current research in the field that could be added to the system.

2.2 Message Layer

Messages exchanged by the Chimera application consist of a message type, a destination key, and the message payload. The type field is set either by the *routing subsystem* or the application. Each message is handed off to an upcall in the routing system which will do any processing necessary and alert the application via upcall if requested. This effectively implements an event-driven message passing architecture, where each message is processed by an upcall at the receiving end. The key field is used only for messages routed through the Chimera routing mesh, and may be left empty for

certain message types. The payload is the part of the message that is delivered to the application.

Chimera communicates using UDP messages sent over BSD sockets. We implement a simple, application layer network protocol on top of UDP to collect feedback on message exchanges. Each network send made by the Chimera application is assigned a monotonically increasing sequence number. This sequence number is prepended to the communication, and the receiver responds with an application level acknowledgement. If this acknowledgement does not arrive within a certain timeout, the send fails. In this case, the system may choose another valid route for the message, or it may reissue the send to the same destination. If the message is acknowledged, the communication time is recorded and sent to the *measurement subsystem*.

The current message system is fairly primitive, and offers many opportunities for improvement. For one, timeouts are currently statically determined, but could be based on measurement information. Additionally, this system could use failures in communication to detect host failures in a more sophisticated way. Also, the system currently treats each send as atomic, and if a send is not acknowledged, it is cancelled and reissued with a new sequence number. By reusing sequence numbers, it could be possible to be more tolerant of hosts whose responses have been slowed by intermittent conditions.

2.3 Host Cache

The host cache is used to maintain information on hosts that might be useful to Chimera in the future. The cache contains two data structures: a searchable list of all hosts in the cache and an LRU list of hosts that are not currently in use by the system. When a host is part of the peer's leaf set, PRR routing table, or application data structure, it is locked into the cache. When all references to the host are removed, it is appended to the end of the free list. When a new host entry is requested, the host looks for it in the cache. If it is found, it is removed from the free list if necessary and returned. Otherwise, a new entry is created. If the cache exceeds its maximum size, the first item is removed from the free list and replaced. Otherwise, a new item is allocated. The cache is allowed to grow past its maximum size if necessary, but it will try not to.

The host entries store both basic host information (name, address, key, port) as well as auxiliary performance information. The main purpose of the cache is to maintain this information even if the host is no longer actively being used. A host that frequently communicates with a given peer will likely have an entry in the cache. If another host fails and the peer needs a new routing entry, it then has a collection of hosts with known expected performance.

The cache is complete as is, but it identifies hosts by name and port only. It would be more powerful if it was capable of searching for hosts based on key.

2.4 Performance Monitoring

The performance monitoring component is responsible for collecting and reporting data on network conditions. This subsystem receives data from the message layer regarding communication times and packet loss. It then uses this information to update the host cache entry regarding the expected performance of link to that host. Other components that access the host entry can then use this information to make decisions based on the expected performance of communication with a given host.

Currently, latency is determined using an exponential average of all communication times seen. The current function gives 10% weight to the last measurement and 90% weight to the measurement history ($m_{n+1} = (0.9 * m_n) + (0.1 * m)$). The loss ratio is computed using the percentage of failed communications. These prediction mechanisms are very simple and inaccurate, and in particular

exponential averages produce a highly varying prediction. There are more effective, but also more complex, methods of predicting network performance.

2.5 Job Queue

In order to limit the number of threads that a program generates and reduce the cost of thread creation, Chimera creates limited number of threads during the initialization phase. All these threads begin as inactive, waiting for jobs to be submitted to the job queue. As jobs are submitted to the pool, these threads wake up and begin to process the job requests. If the number of jobs exceeds the number of threads, they are put in a backlog and are processed in first-come, first-served order.

It might be desirable to process jobs in a different order than simple FCFS. In particular, it might become necessary to processes Chimera routing messages with a higher priority than application messages. It might also be simple to profile different job types and process them using an approximate shortest job first schedule. However, until there appears to be a need for these improvements, they will not be implemented.

2.6 Application Upcalls

Application upcalls are the method through which the Chimera system interacts with the application built on top of it. In response to certain events, such as message delivery, route requests, and leaf set changes, the Chimera system calls functions provided by the user. The user implements applications using these upcalls as described in work on developing a common API for these types of systems [1]. These upcalls are described more completely in section 3.1.

3 API

One of our goals in this project was to provide a flexible API that decouples the user application from the underlying peer-to-peer routing and lookup system. The Chimera API follows the syntax defined in "Towards a Common API for Structured Peer-to-Peer Overlays" [1]. Use of this API eases substitution of underlying routing with minimum effort and facilitates user level application testing based on different routings.

The Chimera interface consists of two main parts: the routing state access and the message upcall interface. The routing state access interface allows an application to directly access to routing state information. These calls can be used to access routing information and make application level routing decisions. The message upcall system interfaces with the application based on the events in routing layer. We will describe the message upcall interfaces and routing state access in detail in sections 3.1 and 3.2.

The functions describe the core interface to the Chimera systems. Other functions to exist to fine tune Chimera performance and behavior, but those functions are left to technical documentation.

3.1 Message Upcalls

All the upcall functions take a function pointer as their only argument. This variable points to a function provided by the application that is declared as described. This system allows the user to respond to events that occur in the routing layer. They also allow the applications to make changes to some of the decisions made by Chimera. There are three upcalls, and they are described here.

typedef void (*chimera_update_upcall_t)(Key *key, ChimeraHost *host, int joined);
void chimera_update(chimera_update_upcall_t func);

The update upcall function is called when a *host* with a given *key* leaves or joins local node's leaf set. The *joined* integer is 0 if the node leaves the leaf set and 1 if it joins the leaf set.

typedef void (*chimera_forward_upcall_t)(Key **key, Message **msg, ChimeraHost **host);
void chimera_forward(chimera_forward_upcall_t func);

The forward upcall informs application that routing layer is about to forward a message *msg* towards the destination *key* via a *host*. This upcall allows the application to modify any of the parameters to override the routing decisions made by the Chimera routing layer. This allows the application to change the next hope, the message, or the destination key.

typedef void (*chimera_deliver_upcall_t)(Key *key, Message *msg);
void chimera_deliver(chimera_deliver_upcall_t func);

This upcall occurs when the current node receives a message *msg* destined for a *key* that is responsible for. This upcall indicates that the message has reached its final destination.

3.2 Routing

This API allows the application to access routing state and pass down application routing preferences. Application layer feedback to the routing layer provides the application with the flexibility to enforce its policy on the underlying peer-to-peer routing and lookup system without needing to modify the routing mechanism.

ChimeraHost **route_lookup(Key key, int count, int is_safe)

This call returns an array of at least *count* ChimeraHost structures that represent acceptable next hops on a route toward a *key*. This version of Chimera ignores *is_safe* and *count* variables, but we keep them in the syntax for future compatibility. The current version of Chimera returns only the best next hop based on routing protocol that we described in section 3.2. If there is no hop closer than the local host, and the message has reached its destination, this function returns NULL.

void chimera_route(Key *key, Message *msg, ChimeraHost *hint)

The chimera_route function will send a message *msg* through the Chimera routing layer to the destination *key*. If the *hint* option is not NULL, the system will use the provided host as its next hop. This call will cause either a chimera_deliver or a chimera_forward upcall, described in section 3.1. This will eventually route the message to the key root in the existing overlay.

ChimeraHost **route_neighbors(int count)

This call will return an array of the *count* closest nodes in the leaf set of the local node. The returned array is NULL terminated, so that if there are not enough hosts in the leaf set to service the request the array will be terminated early.

4 Results

Because the current version of Chimera unstable, there are no performance results to report. The last stable release of Chimera does not support enough of the interesting features to warrant analysis.

5 Conclusion

This paper describes our design of the Chimera peer-to-peer routing system. Although our final version is not complete, we achieved reassuring success working with the Quartz project. Our earliest implementations of the Chimera system supported the upcall and routing interfaces. This allowed the Quartz implementers to design a system using this interface even though the routing layer was significantly simplified. Later stable releases updated the infrastructure without changing the interface, allowing us to transparently modify the implementation.

References

[1] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common api for structured p2p overlays. In *IPTPS*, 2003.

[2] G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *9th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1997.

[3] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. Technical Report UCB//CSD-03-1299, University of California, Berkeley, December 2003.

[4] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.

[5] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.