NChord 1.0.1.0

Technical Documentation and Tutorial

Andrew Cencini

May, 2009

Table of Contents

Introduction

NChord is a C# implementation of the Chord [1] distributed hash table (DHT). This project is designed to provide researchers and developers with a mature and stable .NET version of Chord that may be easily and freely extended and incorporated into all types of applications. Originally developed for a graduate school distributed systems course project, NChord has evolved into a more usable, reliable, and better-organized version than its predecessor.

This document describes how NChord works and may be used, and assumes a modest working knowledge of Chord (reading the Chord paper and understanding the basic Chord protocol, organization and API should be sufficient). In addition, working familiarity with C# and .NET is assumed, though Java and C/C++ developers should have minimal trouble reading and understanding the NChord code and this document.

Requirements

This version of NChord was developed using Visual Studio 2005 and .NET Framework version 2.0.50727 (SP1). Solution and project files are included with the source distribution to aid in viewing, modifying and building NChord via the Visual Studio IDE or on the command line.

Visual Studio 2005 and 2008 should be able to build NChord without major difficulty; building NChord under older unmodified versions of Visual Studio has not been tested. Other versions of the .NET framework have not been tested with NChord though it is likely that newer .NET releases should also support NChord (it is unclear whether older, especially pre-2.0 releases of .NET can support NChord without changes).

Porting

An earlier (but not substantially architecturally different) version of NChord was ported to Mono on Ubuntu Linux and built using Eclipse with minimal difficulty. NChord makes extensive use of .NET remoting; any reasonably recent build of Mono should provide sufficient support for all framework facilities needed by this distribution. Only very limited testing has been performed to gauge interoperability between Mono/Linux and .NET/Windows NChord clients – there appear to be some difficulties, most likely due to serialization differences, which may have since been resolved.

License

NChord is distributed under the MIT license.

Setup & Installation

NChord is distributed in both source and binary form. To make use of the binary distribution, unpack the contents of NChord_bin_1.0.0.0.zip to a project directory, and add a reference to NChordLib.dll to a new or existing Visual Studio project. If using the source distribution, unpack the contents of NChord_src_1.0.0.0.zip to a project directory, and build or load NChord.sln using Visual Studio.

Architecture

NChord is designed to be built into a self-contained library that provides core Chord routing, lookup and ring maintenance functionality. As mentioned earlier, NChord uses .NET remoting for its cross-node communication. .NET remoting was chosen because it is a simple, robust, relatively efficient, and universal communication mechanism, and did not require implementation of any special-purpose networking code.

By relying on .NET remoting, cleaner remote access semantics are provided, where use of a remote object is, in many ways, not much more complicated than accessing an object locally. A static utility class (ChordServer) provides a set of convenient safe wrappers for remote method and property accesses to the remoted object to provide automatic retry logic and simplified exception handling semantics for callers. These wrappers have proven to greatly improve reliability and simplify code by factoring out uniform retry logic and exception handling code that can often fall prey to cut-and-paste errors and contribute to code bloat (and lack of readability).

Forming a Ring of NChord Nodes

Each NChord node is exposed as a singleton instance of an object (ChordInstance). A ChordInstance has a handful of public properties and methods corresponding roughly to those specified in the Chord paper (and discussed in detail later in this document) – each activated ChordInstance listens on a given TCP port for NChord remoting calls. A ring of NChord nodes may be formed by joining ChordInstances together via a seed node (a

node known to already be a member of a Chord ring – the first node in a Chord ring does not have a seed node, however). When participating as part of a Chord ring, nodes communicate with each other (via safe-wrapped remoting calls) in order to perform various routing, data storage and ring maintenance tasks as described in the Chord paper.

See the "Command-Line Server" section later in this document for information about how to use the included NChord sample command-line server application to form and simulate a multi-node NChord ring.

Simulating Multiple Nodes

In addition to using network connectivity to connect NChord nodes resident on multiple physical machines, a Chord ring may also easily be simulated on a single physical machine simply by activating and joining multiple ChordInstances on different TCP ports on the same machine.

Scalability / Ring Size

Using the NChord distribution, a Chord ring of virtually any size may be formed. This particular implementation has been tested on a ring of 229 physical nodes, as well as over a 5004-virtual-node topology spread across roughly 40 physical nodes. From this testing experience, it was obvious that NChord should be able to scale at least 2-3 orders of magnitude beyond these boundaries, which were imposed only on account of hardware resource availability.

Maintenance

Each ChordInstance automatically performs its own maintenance via a set of background worker threads that run while a node is part of a Chord ring. On NChord node Join / Depart, maintenance threads are started / stopped automatically. The NChord maintenance routines run independently, and additional maintenance tasks may be added fairly easily when extending NChord.

Data Storage

One functional area that was intentionally left unimplemented was that of the actual storage and retrieval of data – there are two reasons for this:

1.) Most users will want to provide their own specialized storage semantics (e.g. to support structured storage, indexing, assembly loading, etc.).
2.) Providing a tutorial on how to add the simplest (string key/value) storage and retrieval is an excellent way to familiarize users with the NChord codebase, while also adding basic functionality tied to a core use case of Chord (storage).

A tutorial (including all the necessary code snippets) on how to add simple storage and replication is provided later in this document.

Assumptions & Limitations

A few assumptions were made in implementing NChord:

-64-bit ID space: IDs in NChord are stored as UInt64 – some other implementations use larger or smaller key spaces. It is possible to modify NChord to use a different-sized ID if necessary.

-64-bit truncation of MD5 hash digest: Hash values are generated using MD5, truncated to 64-bits. Truncation is performed in order to align with the ID space limitation. MD5 is provided as the out-of-box hash mechanism; swapping in others (e.g. SHA-1, etc.) is relatively straightforward.

-Remote node identification is hostname-based: As NChord was initially designed for use in controlled experimental environments, network hostname plus TCP port number were originally chosen to serve as ways of identifying and communicating with remote nodes. Broader deployments will likely want to switch to using a serializable variant of IPAddress plus TCP port number.

-No data store: As mentioned earlier, a data store is not provided out-of-the-box, but instead a tutorial demonstrates how to add a working data store to NChord. It is assumed that data storage and retrieval is one of the areas most likely to be customized, therefore, familiarizing NChord users with implementing their own generic data store would be of more value than simply including a minimally useful throwaway store.

-Automatic rejoin: A simple (and extendable) maintenance task to automatically detect and repair ring damage is provided. For Chord purists (or those unhappy with the limited detection/repair semantics), this maintenance task, described later, can be disabled.

-Barebones logging: NChord currently logs to the console. It is assumed that consumers of NChord will replace the built-in NChord logging logic with custom logging (since every application tends to have its own special logging semantics).

-SuccessorCache size: The size of the NChord successor cache is currently hard-coded to three successor cache entries. The successor cache size is easy to change from a single location (ChordInstance.Join.cs) and should be made configurable in later releases.

-Maintenance intervals: Maintenance intervals are currently coded into NChord; configuration through file or API should be made available in later releases.

Project Structure

There are two projects contained in the NChord Visual Studio solution: NChordLib, and NChordServer. NChordLib contains core Chord logic and functionality, as well as the remoting facilities and wrappers – NChordLib builds a DLL (NChordLib.dll) that is referenced by other projects that wish to use NChord's functionality. NChordServer is a sample of a very basic console server implementation that uses NChord. Future releases may include additional sample NChord applications (e.g. NChord running as a Windows service, etc.).

NChordLib

The NChord library is organized across a number of files, where file naming follows the scheme of <className>[.functional_area[.functional_subarea]].cs. File headers provide greater detail on file contents; however, a brief overview is provided here.

ChordFingerTable.cs: contains the ChordFingerTable class implementation that is used as a ChordInstance's finger table.
ChordInstance.cs: constructor and lifetime service management for ChordInstance class.
ChordInstance.Depart.cs: implementation of the Depart functionality for a ChordInstance to depart from an NChord ring.
ChordInstance.Join.cs: implementation of the Join logic for a ChordInstance to join an NChord ring.
ChordInstance.Maintenance.cs: provides the background thread infrastructure to support and control NChord maintenance tasks.
ChordInstance.Maintenance.ReJoin.cs: implementation of the ring partition detection and re-join/re-formation maintenance task (extension beyond standard Chord semantics).
ChordInstance.Maintenance.StabilizePredecessors.cs: implementation of the standard StabilizePredecessors Chord maintenance task.
ChordInstance.Maintenance.StabilizeSuccessors.cs: implementation of a maintenance task that keeps a ChordInstance's successor and successorcache up to date.
ChordInstance.Maintenance.UpdateFingerTable.cs: implementation of the maintenance task used to keep a ChordInstance's finger table up to date.
ChordInstance.Navigation.cs: contains core Chord navigation (routing) logic (FindSuccessor and FindClosestPrecedingFinger).
ChordInstance.Notify.cs: implementation of the Chord standard Notify function used to notify a node of the identity of its predecessor.
ChordInstance.Properties.cs: exposes core public properties of a ChordInstance, used by navigation and maintenance as well as other consuming applications.
ChordNode.cs: contains the ChordNode class, used to identify nodes in an NChord ring.
ChordServer.cs: static ChordServer class containing convenience functions and safe remoting access wrappers used by NChord clients and servers.
ChordServer.Hash.cs: implementation of the hashing logic used in determining a node or key's hash code.

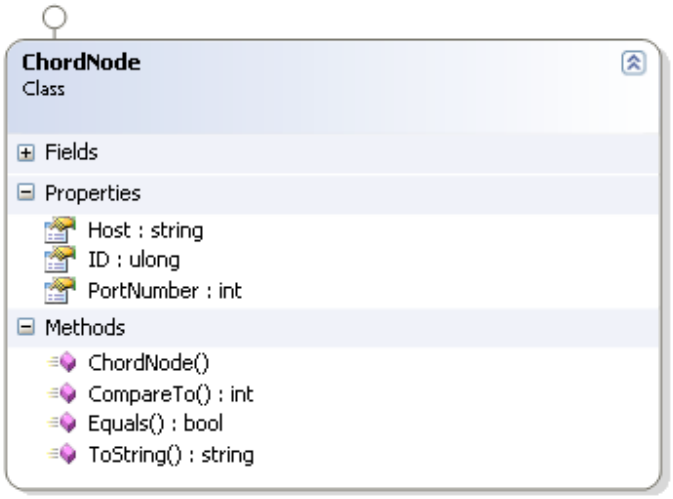
NChordServer

The NChordServer project contains a single-file implementation of a sample NChord console server application. NChordServer is dependent on the NChordLib project for Chord and remoting functionality.

API Reference

The following section describes the NChord API as provided in the NChord source and binary distribution. The NChord codebase is annotated sufficiently to automatically generate API documentation; the documentation provided here aims to provide additional context beyond basic use information.

ChordNode



The ChordNode class is used to provide a structured identification of nodes used in an NChord ring. ChordNode is used extensively throughout the NChord library, specifying a remote (or local) NChord node to communicate with.

ChordNode instances are generally constructed given a hostname (Host) and TCP/IP port number (PortNumber) value. A hash of the host / port combination is generated in order to supply the ID property value. The ID property value is a ChordNode's hash ID in an NChord ring.

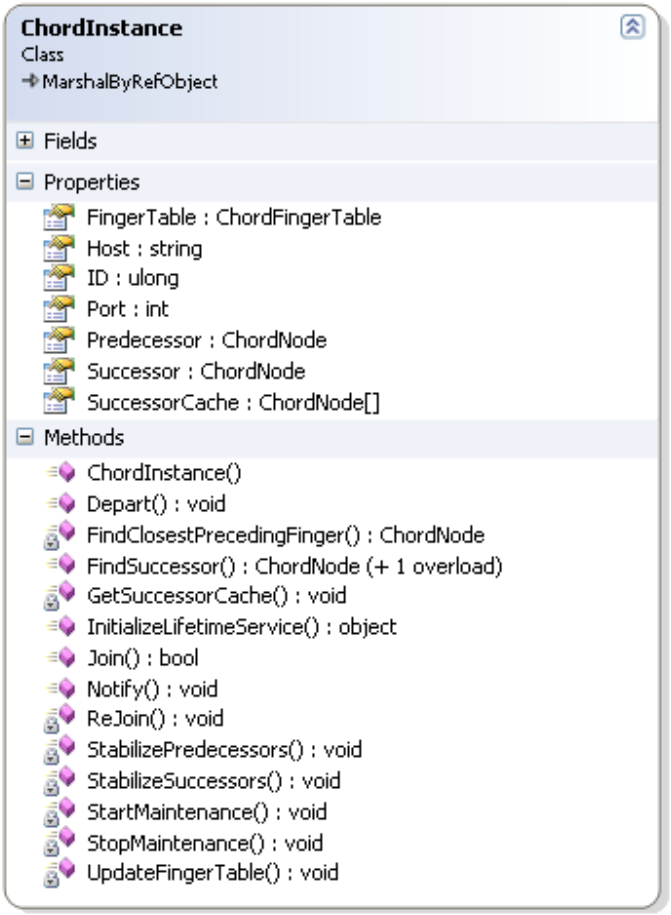
ChordNode object instances implement the IComparable interface, and are compared based on the value of the ChordNode ID property. Similarly, equality operations are also performed on the ID property value of ChordNodes as well.

The ToString() method of the ChordNode object provides a friendly (human-readable) string output of a ChordNode's property values that is useful in logging and debugging.

Note: The ID property of a ChordNode should be compatible with the output of the ChordServer.GetHash() method – in other words, should one desire to change the size of the NChord ID space, those changes should obviously also be reflected in the data type of the ChordNode ID property and internal representation.

ChordInstance

The ChordInstance class contains the main Chord logic. ChordInstance is implemented as a singleton, with an infinite lifetime (similar to other server-style singleton implementations using .NET remoting).



Once a ChordInstance is activated (see ChordServer.GetInstance, below), the various methods and properties of the ChordInstance object become useful.

Join

Typically, a ChordInstance is joined to (or used to start) an NChord ring via the ChordInstance Join method. Join accepts a ChordNode seed node (or null, if the ChordInstance is being used to establish a new NChord ring), as well as a hostname and port number corresponding to the local hostname and TCP/IP port number that the ChordInstance will listen for requests on.

Given the host and port number of the ChordInstance, that instance also automatically receives an ID value (the hash of the hostname and port number) that is used to locate that ChordInstance in an NChord ring.

The seed node supplied on Join is used to determine the new NChord node's successor node (Successor, SuccessorCache), and the ChordInstance maintenance routines take care of updating predecessor information (Predecessor), finger table (FingerTable) entries, and notifying other nodes (via Notify()) in the NChord ring of the presence of the new NChord node.

Depart

An NChord node that is joined to an NChord ring may be removed from that ring gracefully by calling its Depart() method. If an NChord node drops out unexpectedly from an NChord ring, the navigation and maintenance methods are able to respond to the disappearance of that node seamlessly, and the NChord ring is also automatically repaired.

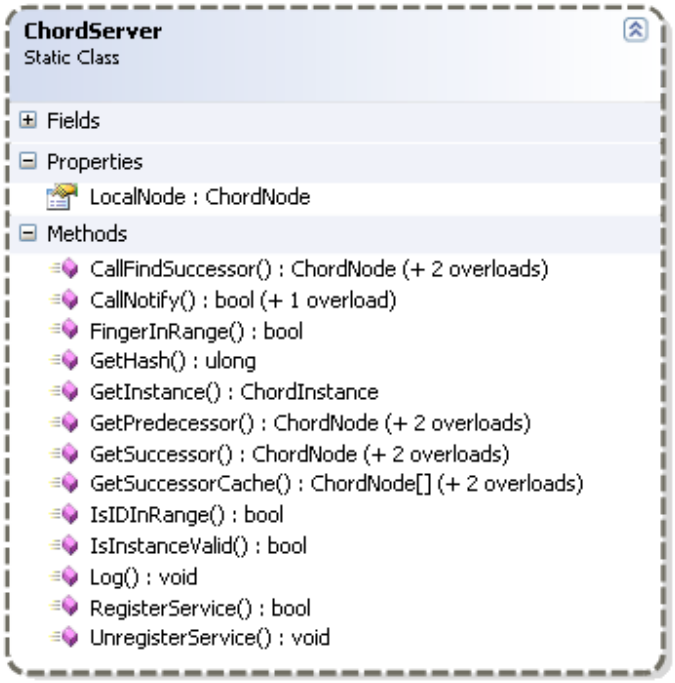
FindSuccessor

The FindSuccessor is the main navigational method used by applications (e.g. custom data stores, etc.). FindSuccessor is used in order to return a ChordNode that is responsible for a given value based on Chord semantics (the responsible node has an ID that is greater than or equal to the specified value and a predecessor whose ID is less than the specified value).

All other members and methods of ChordInstance are used internally for navigation or maintenance purposes.

ChordServer

The ChordServer static class exposes a variety of convenience functions used by NChord client and server applications.



LocalNode

The LocalNode property of ChordServer is used primarily by instances of ChordInstance for self-identification. In future releases, this property may be removed and relocated (e.g. to within ChordInstance itself).

GetInstance

The GetInstance method is used to activate and return a reference to a ChordInstance. A ChordNode object is specified as the sole parameter to GetInstance, identifying the host / port number that the ChordInstance to activate is listening on. If successful, GetInstance returns a ChordInstance whose methods and properties may be directly accessed like any other object; if unsuccessful, null is returned.

IsInstanceValid

IsInstanceValid is often used after calling GetInstance for a given ChordNode. IsInstanceValid takes a ChordInstance object as its only parameter, and returns true if the instance is usable; false, if the ChordInstance is invalid or unusable (e.g. the remote instance is no longer accessible, etc.). Future releases might also extend IsInstanceValid to perform version checking and compatibility between applications and remote ChordInstances.

RegisterService / UnregisterService

RegisterService is used to register a TcpChannel (used for activation of a ChordInstance object) on the local machine, given a TCP/IP port number. If successful, the TcpChannel is registered and true is returned; if unsuccessful, false is returned to the caller.

UnregisterService is used to unregister the TcpChannel registered by RegisterService.

GetHash

The GetHash method returns a hash key for a given (string) value. The GetHash method is central to organization of data in the distributed hash table, and is used by NChord client and server functionality alike.

IsIdInRange

The IsIdInRange method determines whether or not a given key value falls between a start and end value (e.g. local node ID and successor node ID), taking into account wrap-around semantics of an NChord ring.

FingerInRange

The IsFingerInRange method is used by navigation (FindClosestPrecedingFinger) to determine whether or not a given finger in a node's finger table is in the range of a given value, taking into account wrap-around semantics (handles the case where the start and end values are equal differently from the IsIdInRange method).

Wrapped Methods

The following collection of methods all expose a safe wrapper for calling a method on or retrieving a property from a remoting object. At a minimum, a ChordNode object that specifies a remote ChordInstance must be specified (along with any necessary parameters). Overloads to the wrappers also allow retry-count to be specified in order to provide bounding on the number of retries that are attempted when the remote access or call throws an exception.

CallFindSuccessor

CallFindSuccessor is a wrapper used to allow FindSuccessor to be called safely on a given ChordNode to find the Successor node for a given value. If desired, the number of hops taken as part of the FindSuccessor operation may be returned as an output parameter. If the FindSuccessor call fails and exceeds the specified or default retry limit, null is returned; otherwise, a ChordNode object identifying the successor of the specified value is returned.

CallNotify

CallNotify is used internally by ChordInstance's maintenance routines to safely inform a node's successor of its presence. Most applications will not need to use CallNotify.

GetPredecessor

GetPredecessor is a safe wrapper used to retrieve the Predecessor value from a given ChordNode. If the specified ChordNode is unavailable or invalid, null is returned.

GetSucessor

GetSuccessor is a safe wrapper used to retrieve the Successor value from a given ChordNode. If the specified ChordNode is unavailable or invalid, null is returned.

GetSuccessorCache

GetSuccessorCache is a safe wrapper used to retrieve the successor cache from a given ChordNode (e.g. during the StabilizeSuccessors maintenance task). If the specified ChordNode is unavailable or invalid, null is returned.

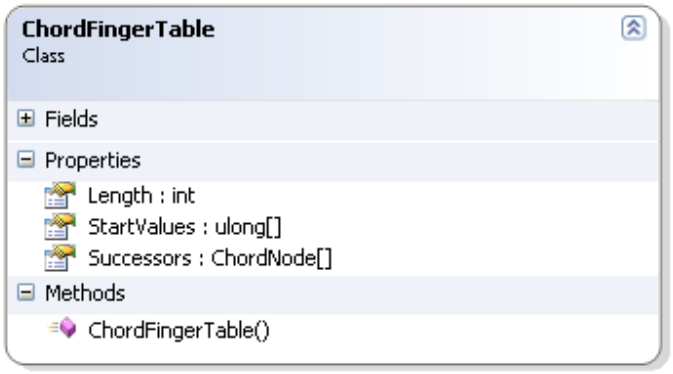
Log Mechanism (Log)

The log mechanism provided with NChord simply logs debug, error, informational, and warning messages to the console. Since different applications tend to employ different logging semantics, it is expected that in many cases the Log method will be re-implemented or forwarded on to an application's own logging facility. Given the pattern of most logging libraries, a log level and log area are both captured by the existing Log method and may be of use to application logging facilities.

LogLevel

The LogLevel enum specifies the log level (severity) for a given log message raised by NChord.

ChordFingerTable



The ChordFingerTable class is used to represent an NChord node's finger table. The length property indicates how many entries are stored in the finger table (in most implementations, typically $\log_2$ of the size of the NChord ID value). Two collections are stored in the finger table – the start values (values exponentially further away from the

local node's ID value) and successors (the ChordNode value that corresponds to the node believed to be the successor for the start value of the same index in the finger table). The ChordFingerTable is constructed and maintained by a ChordInstance in most cases.

Sample Server Applications

Command-Line Server (NChordServer)

A sample command-line server application is provided to demonstrate basic Chord functionality, allowing developers to get an idea of how to establish NChord node instances, connect them together, and display basic ring information. While the code used in the NChordServer sample project may be transplanted into custom applications, there may be more effective, usable or efficient techniques for using and managing NChord instances.

Usage:

First, start an instance of the NChord Server using itself as the seed node (to start a new NChord ring).

system1> NChord.exe <portNumber>

(where <portNumber> is the TCP port number this NChord instance will listen on – e.g. 9001).

Next, start another instance of the NChord Server and specify a valid seed node (e.g. the one started in the previous step).

system2> NChord.exe <portNumber> <seedNodeHostname> <seedNodePortNumber>

(where <portNumber is the TCP port number this NChord instance will listen on – e.g. 9002, and <seedNodeHostname> is the hostname of the seed node – e.g. the hostname of the local node – and <seedNodePortNumber> is the port number specified for the seed node – e.g. 9001). This step can also be used to join to any valid existing Chord Node or ring.

Topographic Map Generator (ChordTopoMapSample)

This is a sample program that can be used to understand how NChord works – both in terms of DHT functionality, and also as a sample program that talks to an existing, running ring of NChord nodes (which can be set up as described in the previous section).

This program assumes that an active, existing ring of NChord nodes exists already. We use the NChord API to talk to nodes in the NChord ring, ultimately generating a map of that ring (comprised of a list of all of the NodeIDs that exist in the ring). Given the map of the ring, a "Topographic Map" (see http://www.cencini.com/ChordTopoMaps) can be

generated by gathering routing hop counts from every node in the ring to every other node in the ring. This data is translated into a JPG image or output as a CSV file.

Windows Service-based Server

In a future release of NChord, a Windows Service-based NChord server sample project will be provided.

Tests

The current release of NChord does not include a test suite; however, an automated test suite will be included in a future release.

Tutorial: Adding Functionality

Add Data Storage

In this tutorial the basic steps to add simple key-value storage are presented. At a high-level, the steps required are:

-Add private member data structure for storing ID, value pairs.
-Add public AddKey method.
-Add public FindKey method.
-(optional) Add wrapped Add/FindKey methods.
-(optional) Add replication maintenance tasks.

First, add a new file (C# class) to the NChordLib project named ChordInstance.Storage.cs. Paste the following template in as the contents of the new file:

```
/*
 * ChordInstance.Storage.cs:
 *
 *  Contains private data structure and public methods for
 *  key-value data storage.
 *
 */

using System;
using System.Collections.Generic;

namespace NChordLib
{
    public partial class ChordInstance : MarshalByRefObject
    {
    }
}
```

The template above will be used to add the private data structure and public methods needed to expose data storage from NChord. Of note are the inclusion of System.Collections.Generic (for the SortedList used as the data store), and the presence

of the partial ChordInstance class (all of the methods and members are being added to the ChordInstance class in this tutorial).

Add Private Data Storage Structure

Next, a data structure must be added to the ChordInstance class to store the key-value pairs. For this tutorial, a simple in-memory SortedList<ulong, string> is used (obviously, other implementations may choose to use different storage semantics).

Add the following code snippet to ChordInstance.Storage.cs:

```
/// <summary>
/// The data structure used to store string data given a
/// 64-bit (ulong) key value.
/// </summary>
private SortedList<ulong, string> m_DataStore =
      new SortedList<ulong, string>();
```

AddKey

Next, add a new method to support adding keys to the data store. Paste the following method into ChordInstance.Storage.cs:

```
        /// <summary>
        /// Add a key to the store.  Gets a hash of the key, determines
        /// the correct owning node, and stores the string value
        /// on that node.
        /// </summary>
        /// <param name="value">The value to add.</param>
        public void AddKey(string value)
        {
            // the key is the hash of the value to
            // add to the store, and determines the
            // owning NChord node
            ulong key = ChordServer.GetHash(value);

            // using the local node, determine the correct owning
            // node for the data to be stored given the key value
            ChordNode owningNode = ChordServer.CallFindSuccessor(key);

            if (owningNode != ChordServer.LocalNode)
            {
                // if this is not the owning node, then call AddKey
                // on the actual owning node
                ChordInstance remoteInstance =
                  ChordServer.GetInstance(owningNode);
                remoteInstance.AddKey(value);
            }
            else
            {
                // if this is the owning node, then add the
                // key to the local data store
                this.m_DataStore.Add(key, value);
            }
```

```
        }
```

The above method implements a very simple way of adding a string-valued key to the data store. Given the string value provided, the hash value of that string (the ulong key) is taken and used to determine the owning node via FindSuccessor called safely on the local node. If the owning node is determined to be the local node (by comparing the static LocalNode property), then the data store (m_DataStore) is used to store the key-value pair; otherwise, the AddKey method is called on the remote owning node by first getting a ChordInstance corresponding to the remote node, and then calling the AddKey method on that instance. In a later step, this sequence will be replaced by a call to a safe wrapper version of the AddKey method.

Note that no locking, duplicate detection, or synchronous replication is performed here on insert. Adding this (and other) functionality is left as an exercise to the user.

FindKey

Next, a method for retrieving values by key is provided. In this case, a ulong key is given by the user, and a string value corresponding to that key is returned (or String.Empty if no such key exists). Paste the following method into ChordInstance.Storage.cs:

```
        /// <summary>
        /// Retrieve the string value for a given ulong
        /// key.
        /// </summary>
        /// <param name="key">The key whose value should be
returned.</param>
        /// <returns>The string value for the given key, or an empty
string if not found.</returns>
        public string FindKey(ulong key)
        {
            // determine the owning node for the key
            ChordNode owningNode = ChordServer.CallFindSuccessor(key);

            if (owningNode != ChordServer.LocalNode)
            {
                // if this is not the owning node, call
                // FindKey on the remote owning node
                ChordInstance remoteInstance =
                  ChordServer.GetInstance(owningNode);
                return remoteInstance.FindKey(key);
            }
            else
            {
                // if this is the owning node, check
                // to see if the key exists in the data store
                if (this.m_DataStore.ContainsKey(key))
                {
                    // if the key exists, return the value
                    return this.m_DataStore[key];
                }
                else
                {
```

```
                    // if the key does not exist, return empty string
                    return string.Empty;
                }
            }
        }
```

The above method works in a fairly similar manner to the AddKey method, except the string value is returned as opposed to stored. (By this point it should be fairly obvious that this particular data store is relatively useless, but the key/value storage can be easily adapted to provide generic object storage for key values generated based on whatever semantics are desired).

Once again, consideration for replicated data stores, locking, etc. is left to the implementer.

Wrapped Methods

Optionally, the above two methods may be added to the ChordServer static class as wrapped methods to allow for safe remote access with basic retry logic included. Add a new file (C# class) named ChordServer.Storage.cs to the NChordLib project, and replace the default contents with the following template:

```
/*
 * ChordServer.Storage.cs:
 *
 *  Exposes wrapped methods used in working with
 *  the sample NChord data store.
 *
 */

using System;

namespace NChordLib
{
    public static partial class ChordServer
    {
    }
}
```

Next, add the following four wrapped methods to the ChordServer.Storage.cs file:

```
        /// <summary>
        /// Calls AddKey() remotely, using a default retry value of
        /// three.
        /// </summary>
        /// <param name="remoteNode">The remote on which to call the
        ///       method.</param>
        /// <param name="value">The string value to add.</param>
        public static void CallAddKey(ChordNode remoteNode,
            string value)
        {
            CallAddKey(remoteNode, value, 3);
        }
```

```
/// <summary>
/// Calls AddKey remotely.
/// </summary>
/// <param name="remoteNode">The remote node on which to call
///       AddKey.</param>
/// <param name="value">The string value to add.</param>
/// <param name="retryCount">The number of retries to
///       attempt.</param>
public static void CallAddKey(ChordNode remoteNode,
    string value, int retryCount)
{
    ChordInstance instance =
          ChordServer.GetInstance(remoteNode);

    try
    {
        instance.AddKey(value);
    }
    catch (System.Exception ex)
    {
        ChordServer.Log(LogLevel.Debug, "Remote Invoker",
          "CallAddKey error: {0}", ex.Message);

        if (retryCount > 0)
        {
            CallAddKey(remoteNode, value, --retryCount);
        }
        else
        {
            ChordServer.Log(LogLevel.Debug, "Remote Invoker",
                "CallAddKey failed - error: {0}", ex.Message);
        }
    }
}

/// <summary>
/// Calls FindKey() remotely, using a default retry
/// value of three.
/// </summary>
/// <param name="remoteNode">The remote on which to call the
///       method.</param>
/// <param name="key">The key to look up.</param>
/// <returns>The value corresponding to the key,
///       or empty string if not found.</returns>
public static string CallFindKey(ChordNode remoteNode,
    ulong key)
{
    return CallFindKey(remoteNode, key, 3);
}

/// <summary>
/// Calls FindKey remotely.
/// </summary>
/// <param name="remoteNode">The remote node on which to call
///       FindKey.</param>
/// <param name="key">The key to look up.</param>
```

```
        /// <param name="retryCount">The number of retries to
        ///       attempt.</param>
        /// <returns>The value corresponding to the key,
        ///       or empty string if not found.</returns>
        public static string CallFindKey(ChordNode remoteNode,
            ulong key, int retryCount)
        {
            ChordInstance instance =
                  ChordServer.GetInstance(remoteNode);

            try
            {
                return instance.FindKey(key);
            }
            catch (System.Exception ex)
            {
                ChordServer.Log(LogLevel.Debug, "Remote Invoker",
                  "CallFindKey error: {0}", ex.Message);

                if (retryCount > 0)
                {
                    return CallFindKey(remoteNode, key, --retryCount);
                }
                else
                {
                    ChordServer.Log(LogLevel.Debug, "Remote Invoker",
                        "CallFindKey failed - error: {0}", ex.Message);
                    return string.Empty;
                }
            }
        }
```

The above methods safely wrap and provide retry logic for AddKey and FindKey. An optional overload uses a default retry value of 3 to permit three retries before the method call attempt is aborted.

Now that the safe wrappers are implemented, a small tweak may be made to the ChordInstance.Storage.cs file to use the wrappers as opposed to raw remote access.

In the AddKey method, replace the following code:

```
// if this is not the owning node, call
// FindKey on the remote owning node
ChordInstance remoteInstance =
      ChordServer.GetInstance(owningNode);
return remoteInstance.FindKey(key);
```

with:

```
// if this is not the owning node, call
// FindKey on the remote owning node
return ChordServer.CallFindKey(owningNode, key);
```

And in the FindKey method, replace the following code:

```
// if this is not the owning node, then call AddKey
// on the actual owning node
ChordInstance remoteInstance =
      ChordServer.GetInstance(owningNode);
remoteInstance.AddKey(value);
```

with:

```
// if this is not the owning node, then call AddKey
// on the actual owning node
ChordServer.CallAddKey(owningNode, value);
```

Maintenance

Finally, very simple replication may be performed by adding a new maintenance task to NChord. First, a simple ReplicateKey method (and wrapper) should be added to ChordInstance.Storage.cs (and ChordServer.Storage.cs):

(ChordInstance.Storage.cs):

```
/// <summary>
/// Add the given key/value replicas to the local store.
/// </summary>
/// <param name="key">The key to replicate.</param>
/// <param name="value">The value to replicate.</param>
public void ReplicateKey(ulong key, string value)
{
    // add the key/value pair to the local
    // data store regardless of ownership
    if (!this.m_DataStore.ContainsKey(key))
    {
        this.m_DataStore.Add(key, value);
    }
}
```

The ReplicateKey method provided above simply accepts a key/value pair regardless of the proper ownership of “key”, and inserts the key/value pair into the local data store if it is not already present. The usage of ReplicateKey is within the ReplicateStorage method (added later in this section), which copies all local keys to the local node’s successor.

(ChordServer.Storage.cs):

```
/// <summary>
/// Calls ReplicateKey() remotely, using a
/// default retry value of three.
/// </summary>
/// <param name="remoteNode">The remote on which to call the
///       method.</param>
/// <param name="key">The key to replicate.</param>
/// <param name="value">The string value to replicate.</param>
public static void CallReplicateKey(ChordNode remoteNode,
```

```
        ulong key, string value)
    {
        CallReplicateKey(remoteNode, key, value, 3);
    }

    /// <summary>
    /// Calls ReplicateKey remotely.
    /// </summary>
    /// <param name="remoteNode">The remote node on which to call
    ///       ReplicateKey.</param>
    /// <param name="key">The key to replicate.</param>
    /// <param name="value">The string value to replicate.</param>
    /// <param name="retryCount">The number of retries to
    ///       attempt.</param>
    public static void CallReplicateKey(ChordNode remoteNode,
        ulong key, string value, int retryCount)
    {
        ChordInstance instance =
              ChordServer.GetInstance(remoteNode);

        try
        {
            instance.ReplicateKey(key, value);
        }
        catch (System.Exception ex)
        {
            ChordServer.Log(LogLevel.Debug, "Remote Invoker",
              "CallReplicateKey error: {0}", ex.Message);

            if (retryCount > 0)
            {
                CallReplicateKey(remoteNode, key, value,
                    --retryCount);
            }
            else
            {
                ChordServer.Log(LogLevel.Debug, "Remote Invoker",
                    "CallReplicateKey failed - error: {0}",
                    ex.Message);
            }
        }
    }
```

The above methods are simply wrappers of the ReplicateKey method, used by the storage replication maintenance task.

Now that the necessary plumbing has been added to permit key/value replication, add a new C# class file to the NChordLib project named ChordInstance.Maintenance.ReplicateStorage.cs. This file should have the following contents, which implement the replication task:

```
/*
 * ChordInstance.Maintenance.ReplicateStorage.cs:
 *
 * Perform extremely simple replication of data store to
```

```
 * successor as a maintenance task.
 *
 */

using System;
using System.ComponentModel;
using System.Threading;

namespace NChordLib
{
    public partial class ChordInstance : MarshalByRefObject
    {
        /// <summary>
        /// Replicate the local data store on a background thread.
        /// </summary>
        /// <param name="sender">The background worker thread this task
        ///       is running on.</param>
        /// <param name="ea">Args (ignored).</param>
        private void ReplicateStorage(object sender,
            DoWorkEventArgs ea)
        {
            BackgroundWorker me = (BackgroundWorker)sender;

            while (!me.CancellationPending)
            {
                try
                {
                    // replicate each key to the successor safely
                    foreach (ulong key in this.m_DataStore.Keys)
                    {
                        // if the key is local (don't copy replicas)
                        if (ChordServer.IsIDInRange(key, this.ID,
                              this.Successor.ID))
                        {
                              ChordServer.CallReplicateKey(
                                    this.Successor, key,
                                          this.m_DataStore[key]);
                        }
                    }
                }
                catch (Exception e)
                {
                    // (overly safe here)
                    ChordServer.Log(LogLevel.Error, "Maintenance",
                        "Error occured during ReplicateStorage ({0})",
                              e.Message);
                }

                // TODO: make this configurable via config file or
                //  passed in as an argument
                Thread.Sleep(30000);
            }
        }
    }
}
```

The above method simply iterates over the keys in the data store, and for each local key (keys that are "owned" by the local node – replicas are skipped), ReplicateKey is called safely on the successor node (if the key is already stored there, then ReplicateKey simply returns). This process, once initialized, sleeps for 30 seconds and then repeats itself.

(Note: this replication scheme is incredibly simple and does not represent a substantially safe or efficient way of replicating data throughout an NChord ring. There are a wide variety of replication schemes listed elsewhere that may be employed, and it is left as an exercise to the implementer to choose or invent a replication scheme that suits his or her own scenario best).

Finally, hook the replication task up to the maintenance facility itself. This may be done by modifying the ChordInstance.Maintenance.cs file to add a new BackgroundWorker to the collection of other BackgroundWorker members:

```
private BackgroundWorker m_ReplicateStorage = new BackgroundWorker();
```

Modify the StartMaintenance method to add the following code:

```
m_ReplicateStorage.DoWork += new
      DoWorkEventHandler(this.ReplicateStorage);
m_ReplicateStorage.WorkerSupportsCancellation = true;
m_ReplicateStorage.RunWorkerAsync();
```

And, finally, add the following:

```
m_ReplicateStorage.CancelAsync();
```

to StopMaintenance().

At this point, a simple, replicated in-memory key/value data store with safe access semantics and automated maintenance has been added to NChordLib. The provided code may be used as a rough guide for adding other NChord functionality – the original and modified files referred to in this section are located under the Documentation\Tutorial\{Original|Modified}\ directory of the NChord distribution.

Contributions and Test Reports

Contributions to the NChord project, as well as experience and test reports are always welcome. To contribute, contact the author or the nchord-users list at the contact information below.

Contact Information

NChord was developed and is maintained by Andrew Cencini (acencini@gmail.com).

For support, bug reports, questions, and discussion, contact nchord-users@lists.sourceforge.net.

NChord web site: http://nchord.sourceforge.net/

Acknowledgments

The author is grateful to all those who have provided support to the NChord project, including Sourceforge.net for hosting the project, and John Frey of Rackable Systems, whose contribution to the structure of the codebase and maintenance apparatus is greatly appreciated.

Revision History

NChord 1.0.0.0 – 8/6/2008 -- Initial NChord Release.

NChord 1.0.0.1 – 5/20/2009 – Minor document update to include usage information for NChord Command Line Server sample application.

NChord 1.0.1.0 – 5/21/2009 – Added documentation surrounding the addition of the new ChordTopoMapSample application.

References

[1] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.* 31, 4 (Oct. 2001), 149-160.