15–610: Engineering Distributed Systems

Project 2: Dynamic Prefetching

Key Dates

Assigned	Monday February 13, 2012
Checkpoint 1	Monday February 20, 2012
Checkpoint 2	Monday February 27, 2012
Due	Friday March 2, 2012

Instructions

- All projects should be implemented individually, not in groups.
- You are free to give and receive help from anyone in class on high-level design issues, clarification on how something works, tracking down hard-to-find bugs, use of Linux tools, structure of the code base, and so on.
- It is *not* acceptable to copy code from/to someone else or tell/ask someone where lines of code have to be added, deleted or modified. If in doubt ask one of the instructors to help you, or check whether what you propose to do is acceptable.

Goal of the Project

The goal of this project is to implement a dynamic prefetching mechanism in Coda. The actual semantics of the prefetch is different for file and directory objects and this is explained later.

This project is divided into three parts. For checkpoint 1, you will add a new command to notify the Coda client (venus) that prefetching is desired. The actual implementation of a simple prefetching algorithm is not necessary until the second checkpoint. For the second checkpoint you are also asked to propose modifications to improve the implementation. Finally you will implement the proposed modifications and evaluate their effect.

After this project, you will understand in more detail the benefits of prefetching. You will also learn that even though prefetching can significantly improve performance, it should not be used recklessly. In particular, there are many situations where careless use of prefetching can cause severe performance degradation.

By the time you successfully complete this project, you will have done the following:

- Gain hands-on familiarity with the Coda Distributed File System.
- Modified cfs to enable, disable, and control prefetching
- Modified Venus to implement a simple version of dynamic prefetching
- Demonstrated situations where dynamic prefetching is beneficial as well as situations where it is harmful

- Refined the implementation to make prefetch smarter
- Demonstrated that there are cases where the refined implementation improves performance relative to the simple implementation

Modifying cfs

The commands you will add are:

```
cfs prefetch on [-depth <n>] [-contentalso]
cfs prefetch off
```

`cfs prefetch on` will be used to enable the dynamic prefetching and `cfs prefetch off` will clearly be used to disable prefetching.

At this point we are not yet going to be concerned with implementing an actual prefetch mechanism. However you will have to implement the necessary parts so that the cfs process uses the pioctl (path-based ioctl) mechanism to send a message through the kernel module to the Coda client process.

Implementation hints

The code for the cfs command is found at `coda-src/vtools/cfs.cc`. It uses pioctls to forward user requests to the Coda client process. These pioctls operate mostly identical to normal UNIX ioctl which are used pass special commands to in-kernel device nodes. The main difference is that ioctl only works on character and block devices and in a limited way on files, because it relies on an open file descriptor. However we needed this functionality to work also on directories and symlinks, so instead of a file descriptor, we pass a path as the first argument.

Internally pioctls are implemented as an ioctl operation on a special and invisible `/coda/.CONTROL` file that only exists in the Coda kernel module. The kernel then maps the path to a file identifier and passes the message to the Coda client.

The Coda client (`coda-src/venus/*`) has a main loop that waits for messages from the kernel module which are decoded and handed of to a worker thread (`worker.cc`) which then calls into the pioctl specific handling code.

Checkpoint 1: Demo the cfs prefetch command [20 points]

When you run `cfs prefetch`, the Coda client should log a message showing that it has received the request and what the *-depth* and *-contentalso* arguments were.

Implement simple dynamic prefetching in the Coda client

As specified in the checkpoint 1 description of the new prefetch call, there are two optional arguments for `cfs prefetch on`.

-contentalso	prefetch the content of files/symlinks whose attributes are accessed.
-depth x	prefetch attributes for the children for all directories levels below a directory that was demand-fetched. The default should be 1.

For example, the default value of 0 tells the client to prefetch the attributes of all children of the directory whose contents is accessed. A value of N, where N $>$ 0, should trigger a prefetch for the content of any

subdirectories and perform those fetches with a effective depth set to N–1, and as such will recursively retrieve the attributes of their children. `cfs prefetch off` tells the Coda client to stop prefetching. Any already queued prefetch requests should also be discarded. When we access an object in the cache, the request can be for attributes (also referred to as metadata or status) or for content (which is also referred to as data).

Prefetch decisions can be broken up into four cases:

1. File or symlink attributes are accessed and *-contentalso* was specified, prefetch contents.
2. File or symlink data is accessed, there is nothing that needs to be done.
3. Directory data is accessed and prefetching is active (depth >= 0), Venus should prefetch attributes for all objects located in the directory.
4. Directory attributes are accessed, we may have to prefetch the directory contents if the depth is non-zero.

Implementation hints

It is best to implement prefetching as a separate daemon thread in Venus. Look at the existing Hoard daemon in `coda-src/venus/hdb_daemon.cc` for an example of how to create a separate thread in Coda. In particular, look at the *HDBD_Init* function to understand how to start a thread. The *HDBD_Request* function shows how threads in the system can use a message queue to send requests to the daemon. Finally the *HDBDaemon* and *HDBD_HandleRequests* functions show how a daemon thread receives messages from a queue and processes them.

Venus has a function *fsdb::Get* in `coda-src/venus/fso0.cc` that is used to obtain either the metadata or the data. Fetching the data will also fetch the metadata if it wasn't already cached. A good way to implement prefetching is to extend this Get function. Unfortunately, *fsdb::Get* is quite long and fairly hard to understand, it has to handle quite a few error conditions. Fortunately, you only need to issue prefetch requests after the object has been successfully retrieved which is at the end of the function around the same place where Venus updates the object priority.

Do not pass a pointer to the *fsobj* in prefetch request messages, but store the object identifier (*VenusFid*), then the prefetch thread can use *fsdb::Find* to re-obtain a reference to the *fsobj*. This way the object can be safely discarded when it received an invalidation callback or because of cache pressure. Also, do not to forget to release objects with *fsdb::Put*.

When iterating through a directory, do not use the *children* and *child_link* in the *fsobj*, these are local caches and will be empty after we've fetched a new directory object. Instead you want to access the *PDirHandle* at *data.dir->dh*. Use this in conjunction with *DH_EnumerateDir()* to walk over the directory contents. When converting from directory entries to fsobjs, you will need to use *FID_NFid2Int()*.

Checkpoint 2: Demo the simple prefetching mechanism [40 points]

- Show that prefetching can result in performance improvements.
- Show situations where prefetching is actually harmful. In particular, think about the effect of prefetching when bandwidth is limited or the cache is too small.
- Show situations where *-contentsalso* and *-depth* are useful.
- Submit a short writeup proposing how you can improve prefetching.

Implement an improved pretching mechanism

Improve the prefetching mechanism and evaluate the effectiveness of the improvements.

A very important factor when evaluating the effectiveness of prefetching is the cache advantage. If the cache advantage is low it will be very difficult to show improvements from the prefetch implementation. However in our case we can quite easily, and legitimately, increase the cache advantage by reducing available network bandwidth and increasing round trip time or latency. The next page documents various ways in which we can artificially introduce simulated network characteristics and make the Coda client believe it is connected to the servers via a dialup modem or GSM connection.

It should also be noted that both prefetching and demand fetching use the same finite resource, available network bandwidth. As a result prefetch activity will interfere with demand fetches that are needed to handle cache misses.

Final: Submit and demo the improved mechanism [40 points]

- Submit your code:
  - Make sure any newly added files have been added to the repository with `git add <filename>` and that all changes have been committed with `git commit`.
  - You can check what your commit history looks like by running `gitk` and for any uncommitted state with `git status`.
  - Submit your work by pushing the currently checked out branch,

```
git push git@coda.cs.cmu.edu:coda.git HEAD:project2
```

- Show situations where the smarter implementation outperforms the simple implementation. Explain how your improvements are responsible for the differences.
- Show situations where the smarter implementation performs worse than the simple implementation. Explain why there was a performance loss.

Tutorial: Artificially Controlling Bandwidth and Delay

In many situations, it may be necessary or helpful to test your code under different bandwidth and delay conditions. There are several techniques to control a system's network bandwidth and delay.

Every method has advantages and drawbacks.

One method is to place a machine with a network emulation package like NISTnet on the network in between the client and server. This gives very good control, but it requires quite a bit of setup and configuration and a good connection between the client, network emulator, and the server to obtain reliable figures. We will not be using this, but if you ever have to publish a paper this is probably the preferred method to get a controlled test environment.

The second method utilizes the queuing disciplines in the Linux kernel. An advantage is that these are available on pretty much all Linux systems and it is easy to control the egress (outgoing) bandwidth. However it is difficult to control ingress (incoming) bandwidth. Also several of the available queuing disciplines work only reliably for TCP as their main action is to drop packets. TCP interprets packet loss as network congestion and will automatically slow down it's transmission rate by reducing the size of its send window.

The method we will be using is Coda specific, we will use a Lua script to delay both transmission and reception of Coda's RPC2 requests and responses. The RPC2 library needs to be rebuilt to include the Lua interpreter. This was already done in your source tree when it was configured, if at any point you need

to re-execute configure and want to make sure the Lua bindings are available then make sure you add the *—with-lua* option, i.e. `./configure --prefix=/usr --with-lua`.

If the Lua bindings are active the applications will periodically look for the existence of a script file in `/etc/rpc2.lua`. This file is expected to declare a couple of functions that are called whenever a packet was received or is about to be sent. These functions decide if the given packet should be dropped (*return nil*), passed through (*return 0*) or delayed (any positive, non-zero return value is interpreted as delay in milliseconds). When the file is updated it will automatically be reloaded, if the script cannot be read or is removed the delays are disabled.

You can copy the example script from `/usr/share/rpc2/rpc2-fail.lua` to `/etc/rpc2.lua` to control the bandwidth and latency for Coda applications. The example script is fairly simple and assumes symmetric latency and bandwidth on both transmit and receive path, but it should not be too hard to adapt it for various other scenarios.

Any *print* statements in the script result in a write to *stderr*, which in the case of venus will result in a log entry in `/var/log/coda/venus.err`. The example script logs the amount of time it delayed each packet.

Connection type	bandwidth	latency	packet loss
GPRS cellular modem[a]	32000 — 80000 bits/s	0.35 seconds	0.01%
56k PPP dialup modem	recv 53300, send 48000 bits/s	0.1 seconds	0.001%
802.11b/g wireless	11000000 — 54000000 bits/s	0.0075 seconds	0.001%

[a] These numbers are a rough estimate, in reality things are considerably more complicated. Bandwidth depends on the distance from the cell tower, there is typically a higher download bandwidth. And as bandwidth increases GPRS uses less error correction and will see a higher packet loss. For more details see `http://en.wikipedia.org/wiki/GPRS`.

Table 1: Example bandwidth and latency values. A list of other bandwidth figures is at `http://en.wikipedia.org/wiki/List_of_device_bandwidths`

Tutorial: How to Build and Install Coda

Check out a copy of the Coda source code and set it up for building as follows,

```
sudo apt-get install -y gitk build-essential automake libtool flex bison
sudo apt-get install -y libreadline-dev libncurses5-dev liblua5.1-dev

cd ~
# copy your ssh id_rsa and id_rsa.pub files to ~/.ssh
# or generate a new keypair and send me the public key
git clone git@coda.cs.cmu.edu:coda.git
cd coda
git submodule init
git submodule update
./bootstrap.sh
./configure --prefix=/usr --with-lua
```

You can then build the source tree as follows,

```
cd ~/coda
make
sudo make install
```

When you make changes to the Makefiles, make sure you update only the Makefile.am template from which both the Makefile.in and Makefile files are generated otherwise you might lose changes when the build tree is reconfigured which can happen automatically during a build.

Next we have to update the configuration and create some important directories to store logs and local copies of cached files,

```
sudo codaconfedit venus.conf isr 1
sudo mkdir /var/log/coda /var/lib/coda /coda
```

You should now be able to start the client as follows,

```
sudo modprobe coda && sudo /usr/sbin/venus -init
```

If the client started successfully we will be able to access the testserver in `/coda`,

```
ls /coda/testserver.coda.cs.cmu.edu
```

and see contents on the server. You should be able to navigate around and read various files there. To also modify files you will need to be authenticated.

```
clog guest@testserver.coda.cs.cmu.edu
password: guest
```

You will now be able to create new and modify existing files in the playground directory.

Note: This server is not meant for serious data storage. It is merely a convenience for people anywhere on the Internet to test their clients. Don't use it for any serious purpose. Your bits may vanish at any time!

The Coda client provides different ways of logging activity, the main logs are in `/var/log/coda/` where we find both `venus.log` and `venus.err`. However there is also a non-persistent logging mechanism that gives a lot of information and insight about what is going on in the system, this can be viewed by running the `codacon` command in a terminal.

To orderly shutdown a running Coda client you would run the following,

```
sudo umount /coda && sudo kill `cat /var/run/coda-client.pid`
```

(or killall venus lt-venus). There is an Ubuntu/Debian init script that simplifies starting and stopping the Coda client, it is not installed by default but can easily be copied to the right place manually.

```
cd ~/coda
sudo cp debian/coda-client.init.d /etc/init.d/coda-client
sudo chmod 755 /etc/init.d/coda-client
```

Now starting, stopping and restarting the client can be done with,

```
sudo service coda-client start|stop|restart
```