Tutorial of PA3

Nan Zhu

2013-11

Outline

- Goal of PA3
- Background knowledge preparation
  - Why Distributed File System
  - How Distributed File System Works
  - How Processes Work in Network
- Test Case Analysis

Sorry, small fixes

A fixed version has been uploaded, if you download now, you don't need to fix manually

- Read the discussion board, there is a post pinned on the top
  - 1. In dfs_common.c, in create_client_tcp_socket, you are supposed to return the socket file descriptor, instead of 0 - success, 1 - fail,
  - 2. In dfs_datanode.c, heartbeat(), replace "create a new thread" with "create a new socket"
  - 3. replace test case 7 with
    - int test_case_7(char **argv)
    - {
      - char **args = (char **) malloc(sizeof(char *) * 3);
      - args[0] = argv[0];
      - args[1] = argv[1];
      - args[2] = "local_file";
      - int r2 = test_case_2(argv, 1024);
      - args[2] = "local_file_medium";
      - int r4 = test_case_4(argv, 4096);
      - args[2] = "local_file_large";
      - int r6 = test_case_6(argv, 8192);
      - return r2 || r4 || r6;
      - }
  - 4. In dfs_datanode.c in int mainLoop() assert (server_socket == INVALID_SOCKET) should be assert (server_socket != INVALID_SOCKET)

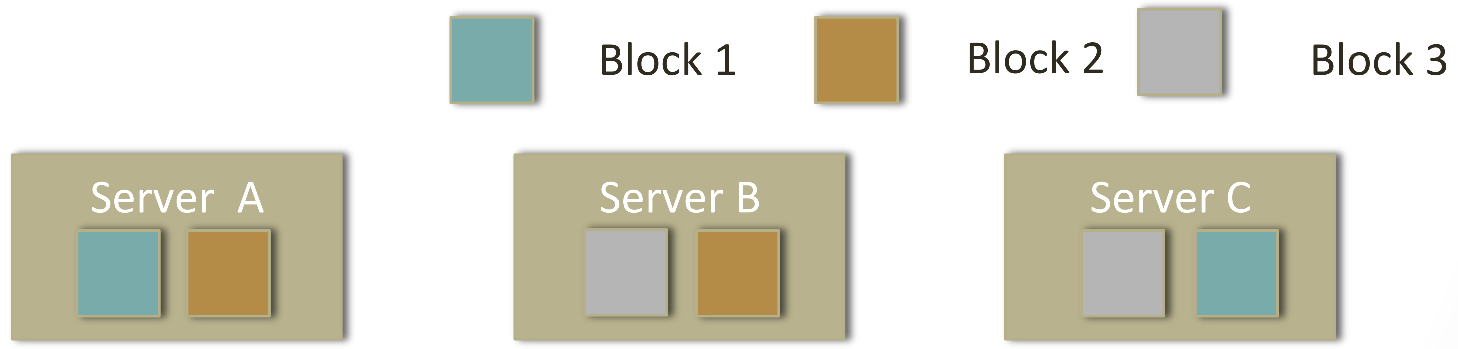
Goal of PA3

- Implement the Distributed File System
  - Inter-processes communication
    - Socket programming
  - Multi-thread Programming
    - POSIX Thread Model

Background Knowledge

Why Distributed File System

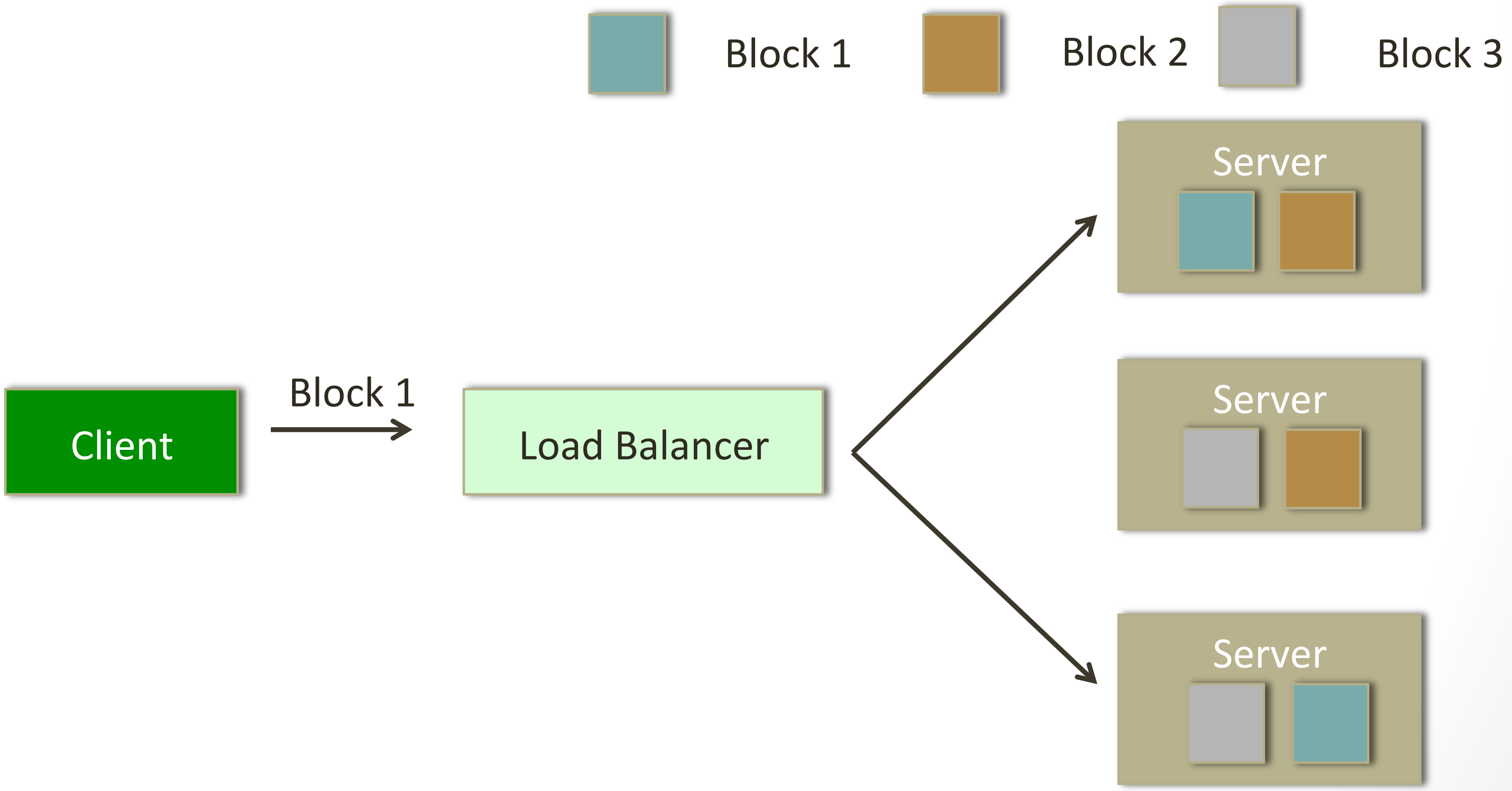
- Increase Capacity
  - Distributed File System coordinates the machines and provide a unified abstraction to the client (the client has not idea about the distributed)
    - The storage capacity in your DFS would be N * disk size, N is the number of storage servers in your DFS
- Fault-Tolerance



A file may consists of Block, 1, 2, 3

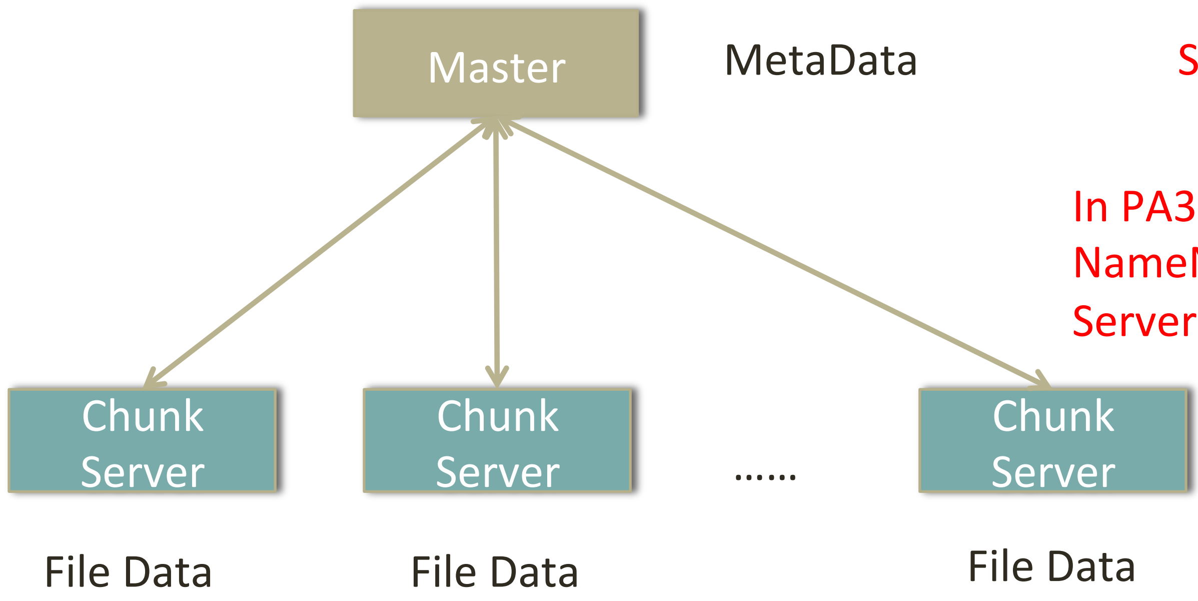
Why Distributed File System

- Improve Performance
  - Load Balance



How Distributed File System Works

- Centralized File System
  - Example: Google File System (GFS), Hadoop Distributed File System (HDFS), etc
  - Easy to implement, comparing to the decentralized version
  - Client always communicates with the system via Master node



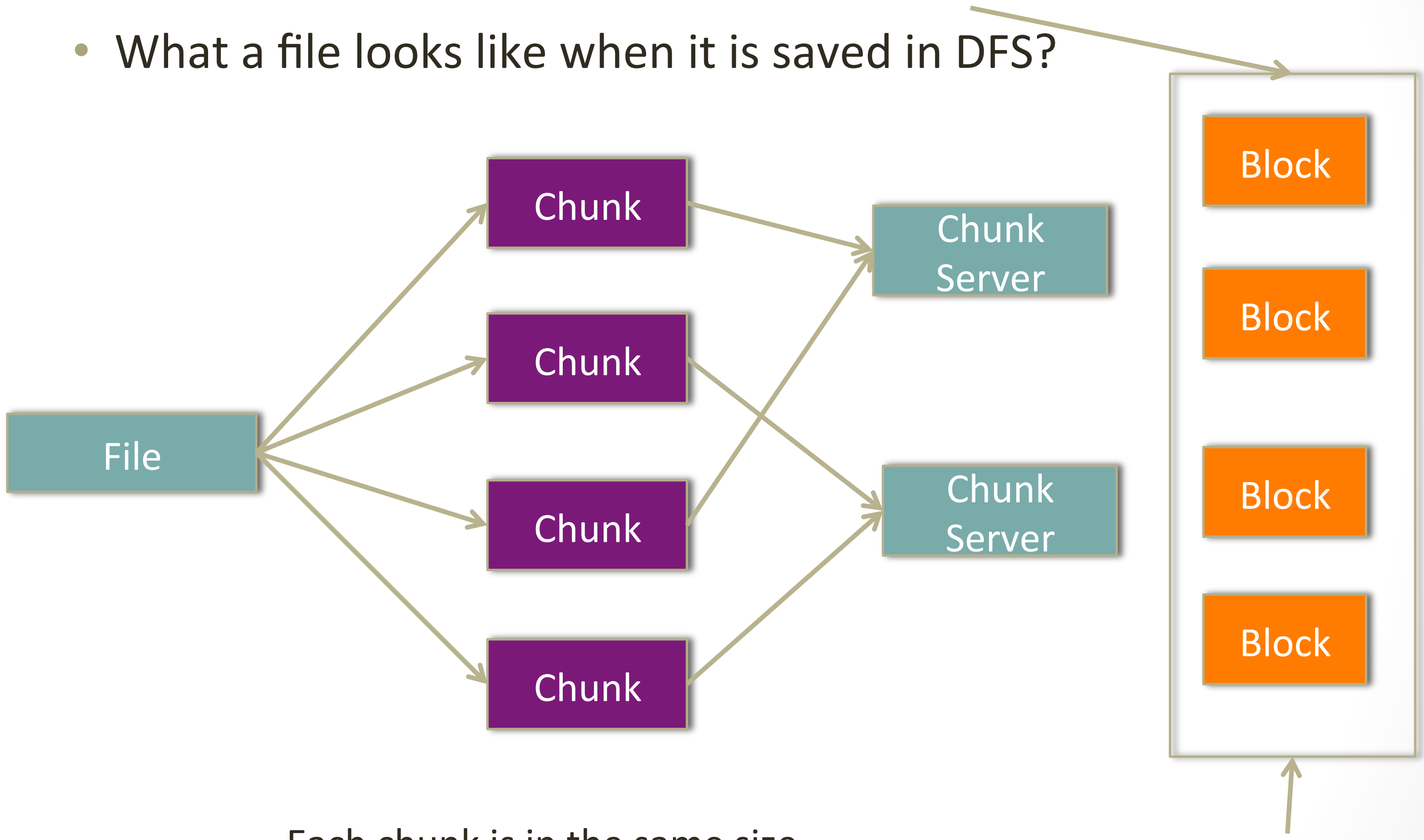
Superblock, inode

In PA3, We call Master as NameNode, and Chunk Server as DataNode

How Distributed File System Works

Each chunk is an independent file in ChunkServer and it is divided into multiple blocks

- What a file looks like when it is saved in DFS?



Each chunk is in the same size
In PA3, we call it DFS_BLOCK, the size is set by DFS_BLOCK_SIZE

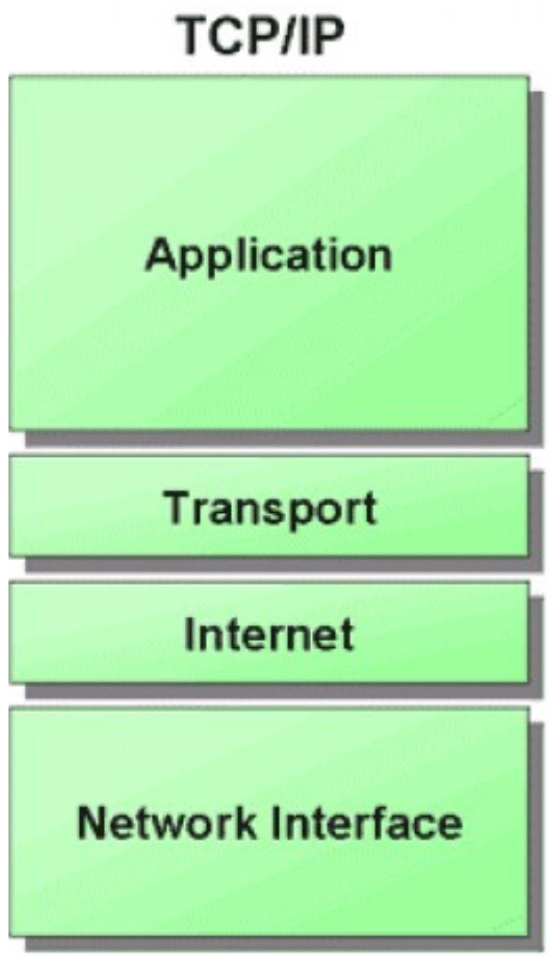
Chunk server's local file system (PA2)

How Distributed File System Works

- Example
  - DFS_BLOCK_SIZE (Chunk size) = 1024 bytes
  - Two chunk servers
  - File1 size = 2048 bytes
  - In ChunkServer's file system, BLOCK_SIZE = 512 bytes
  - File1 -> File1_Chunk1 (1024 bytes), File1_Chunk2 (1024 bytes)
    - File1_Chunk1 and File1_Chunk2 are treated as a file by ChunkServer's local file system and further divided into blocks to save

How Processes Work in Network

- TCP/IP 4-layer Model



The application layer contains the higher-level protocols used by most applications for providing user services over a network and for some basic network support services.[e.g. HTTP]

The transport layer establishes a basic data channel that an application uses in its task-specific data exchange.

The internet layer has the responsibility of sending packets across potentially multiple networks.

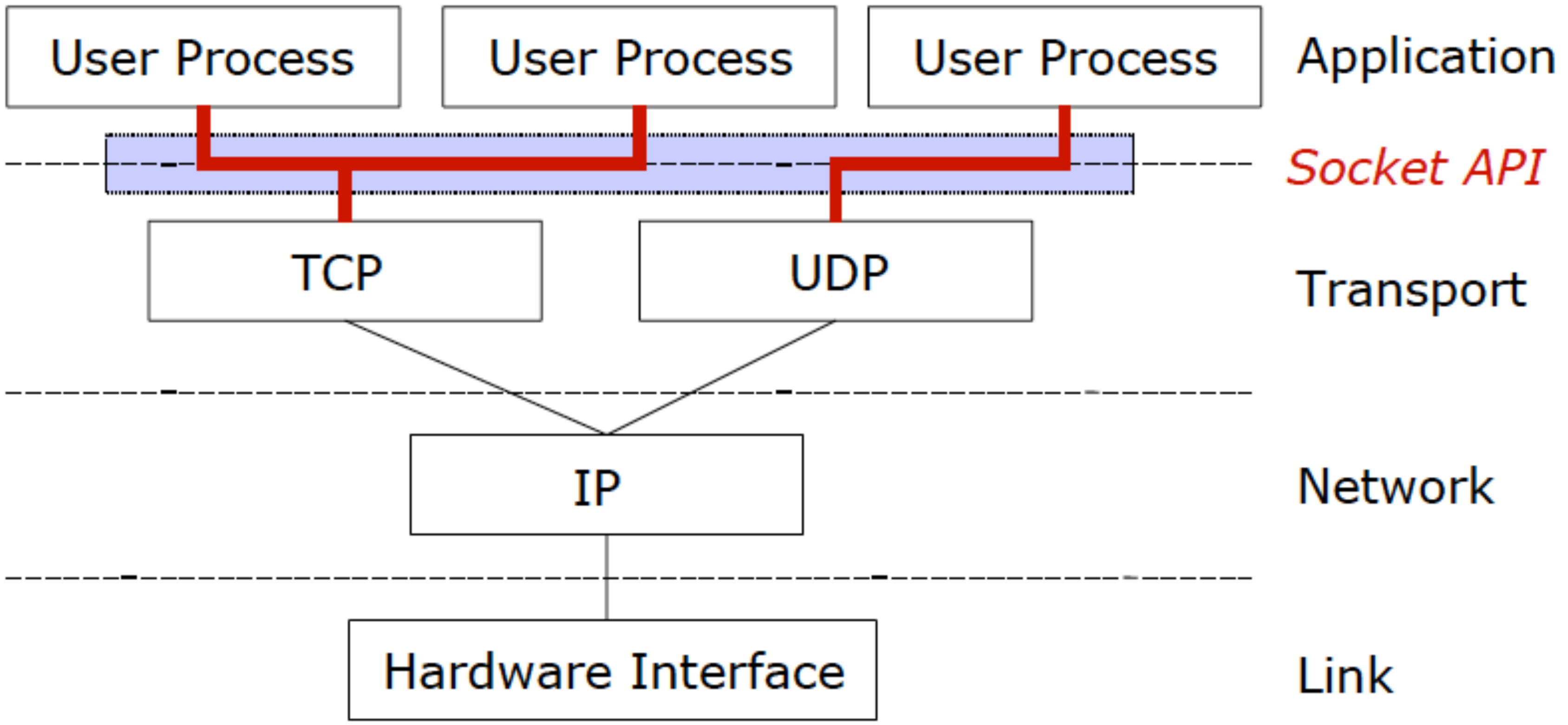
The link layer is the networking scope of the local network connection to which a host is attached.

How Processes Work in Network

- Socket API
  - A collection of system calls to write a networking program at user-level.
  - API is similar to Unix file I/O in many respects: open, close, read, write.
    - Data written into socket on one host can be read out of socket on other host
  - Difference: networking has notion of client and server.
    - Example:
      - Client sends data to DataNode to save
        - Client is client, DataNode is server
      - DataNode sends heartbeat to NameNode
        - DataNode is the client, NameNode is the server

How Processes Work in Network

- Socket API Layer



How Processes Work In Network

- A socket connection has 5 general parameters:
  - The protocol
    - Example: TCP, UDP etc.
  - The local and remote address
    - Example: 171.64.64.64
  - The local and remote port number
    - Needed to determine to which application packets are delivered . Some ports are reserved (e.g. 80 for HTTP- see /etc/services) Root access require to listen on port numbers below 1024

Socket Programming

- Selecting Protocol
  - Connection oriented (streams, e.g. TCP)
    - Congestion control, order guarantee
    - sd = socket(PF_INET, SOCK_STREAM, 0);
  - Connectionless (datagrams):
    - Best-effort sending, random path
    - sd = socket(PF_INET, SOCK_DGRAM, 0);
  - socket() returns a socket descriptor, an int similar to a file descriptor.

Socket Programming

- Selecting Remote Address and Port
  - Client
    - Use connect() on a socket that was previously created using socket():
      - err = connect(int sd, struct sockaddr*, socklen_t addrlen);
    - Remote address and port are in struct sockaddr:
      - struct sockaddr_in {
      - u_short sa_family; (the protocol family you will use in your socket, always set it as AF_INET)
      - u_short sin_port; **(the port number);**
      - struct in_addr sin_addr; (IP address in network order)
      - char sin_zero[8]; //padding
      - };

Socket Programming

- Server
  - if (bind(sockfd, (struct sockaddr *) &serv_addr,
  -                       sizeof(serv_addr)) < 0)
  -    {
  -        perror("ERROR on binding");
  -        exit(1);
  -    }

  -    /* Now start listening for the clients, here process will
  -    * go in sleep mode and will wait for the incoming connection
  -    */
  -    listen(sockfd,5);
  -    clilen = sizeof(cli_addr);

  -    /* Accept actual connection from the client */
  -    newsockfd = accept(sockfd, (struct sockaddr *)&cli_addr,
  -                                &clilen);

Socket Programming

- Client
  - socket() create the socket descriptor
  - connect() connect to the remote server.
  - read(),write() communicate with the server
  - close() end communication by closing socket descriptor
- Server
  - socket() create the socket descriptor
  - bind() associate the local address
  - listen() wait for incoming connections from clients
  - accept() accept incoming connection
  - read(),write() communicate with client
  - close() close the socket descriptor

Socket Programming

- Server Operation
  - The socket returned by accept() is not the same socket that the server was listening on!
  - A new socket, bound to a random port number, is created to handle the connection
  - New socket should be closed when done with communication
  - Initial socket remains open, can still accept more connections

Socket Programming

- Supporting Functions
  - gethostbyname() get address for given host name (e.g. 171.64.64.64 for name "cs.stanford.edu");
  - getservbyname() get port and protocol for a given service e.g. ftp, http (e.g. "http" is port 80, TCP)
  - getsockname() get local address and local port of a socket
  - getpeername() get remote address and remote port of a socket

Socket Programming

- Byte Order
  - Order of bytes in a 32-bit word depends on hardware architecture
  - Big-endian vs little-endian
    - "Little Endian" means that the low-order byte of the number is stored in memory at the lowest address, and the high-order byte at the highest address. (Intel, etc)
      - ABCD, Memory starts from 1000
        - 1000 – CD, 1001 - AB
    - "Big Endian" means that the high-order byte of the number is stored in memory at the lowest address, and the low-order byte at the highest address. (RISC processors)
      - ABCD Memory starts from 1000
        - 1000 – AB, 1001 - CD
  - Network byte order used for portability
    - The Internet Protocol defines big-endian as the standard network byte order used for all numeric values in the packet headers and by many higher level protocols and file formats that are designed for use over IP.

Socket Programming

- Byte Order (2)
  - Use appropriate functions in your assignment htonl(), htons(), ntohl(), ntohs()
    - htonl - converts the unsigned integer hostlong from host byte order to network byte order
    - htons - converts the unsigned short integer hostshort from host byte order to network byte order
    - ntohl - converts the unsigned integer netlong from network byte order to host byte order
    - ntohs - converts the unsigned short integer netshort from network byte order to host byte order.
    - Don't forget to use them for connect() etc.

Socket Programming

- Address conversion routines
  - Convert between system's representation of IP addresses and readable strings (e.g. "171.64.64.64")
    - unsigned long inet_addr(char* str);
    - char * inet_ntoa(struct in_addr inaddr);
- Important header files:
  - <sys/types.h>, <sys/socket.h>, <netinet/in.h>, <arpa/inet.h>

Test Case Analysis

Programming Assignment 3

- Test 0 – Test Heartbeat service
  - Heartbeat service is used to maintain the connection between datanodes and namenode (chunk servers and master)
    - You have to start a new thread in namenode and datanode
      - accept is a blocking operation
  - In dfs_namenode.c
    - dfs_datanode_t* dnlist[MAX_DATANODE_NUM];
      - This array maintains the available datanode information in the system (we don't consider datanode failure)
    - dfs_datanode_t
      - typedef struct _datanode_descriptor_
      - {
      -         int dn_id;
      -         char ip[16];
      -         int port;
      -         unsigned int live_marks;
      - }dfs_datanode_t;

Programming Assignment 3

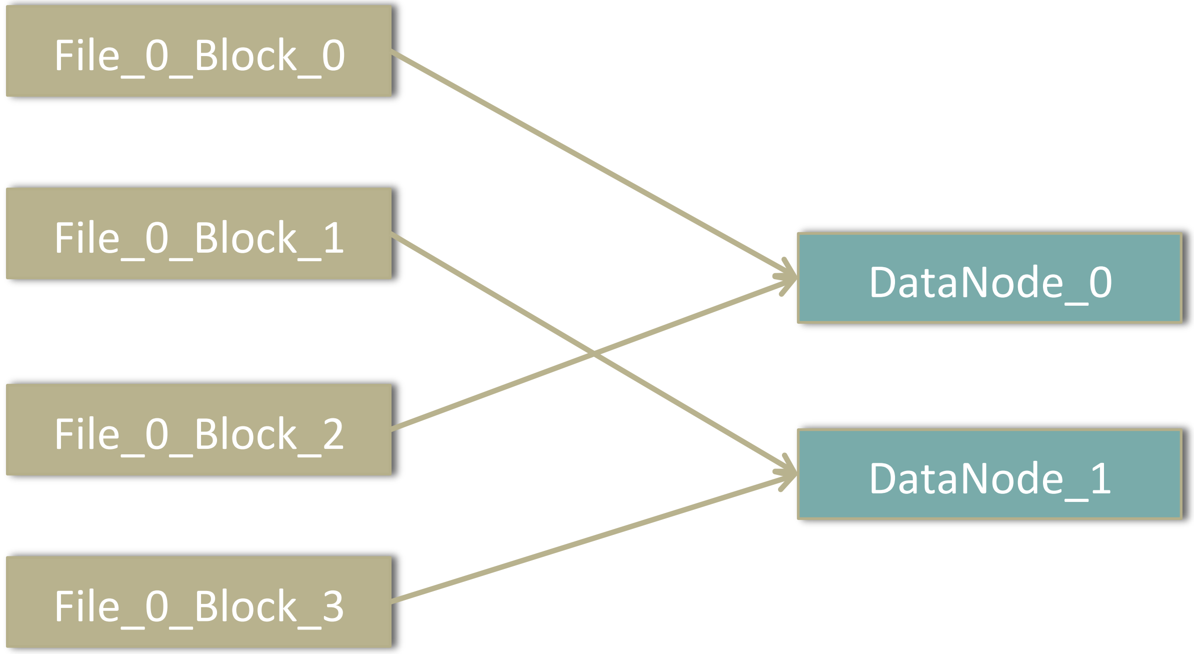
- Test case 0
  - Dfs_namenode.c, start(), create a new thread, the thread handler is supposed to be HeartbeatService(), creates a new socket to listen for heartbeat requests from datanode
  - dfs_datanode.c, start(), create a new thread to report heartbeat to namenode, the handler is heartbeat()
  - Dfs_datanode.c, heartbeat(), create a new socket to send dfs_cm_datanode_status_t datanode_status to the namenode

Programming Assignment 3

- Test 1 – 6
  - Read/Write service in your distributed file system
  - **Client** always communicates with namenode first when it invokes a write/read request
  - For write request, **NameNode** takes the file size as input, divides the bytes into chunks (DFS_BLOCK) and assigns them to the chunk server (**DataNode**). Returns the DataNodes' IP address, port number and corresponding block number to the client
  - For read request, **NameNode** checks its metadata structure (like the inode in PA2), find the required file name and the location of its chunks, returns to client
  - Client sends request to DataNode
  - DataNodes saves or reads data for clients

Programming Assignment 3

- Test 1 – 6 (cont.)
  - Round-robin-like



Programming Assignment 3

- The pairs of Test case 1&2, 3&4, 5&6 are similar
  - 1, 3, 5
    - In test.c, generate_data is called to generate a file containing random data
    - In test_1, 3, 5, send_file_request is called to send a write request is to the namenode
    - dfs_client.c, push_file is called to send the request via socket
    - dfs_namenode.c, mainLoop(), accepts requests from client and calls the handler function (the handler functions are given)
    - dfs_namnode.c, int get_file_receivers(int client_socket, dfs_cm_client_req_t request), sent location information of all datanodes via socket to the client
    - dfs_client.c, in push_file(), client receives location information and send the data to the datanodes,
    - dfs_datanode.c, mainloop(), accept request from client and calls the handler function to create data locally (the handler functions are given)
    - Test.c compares the original file and the data saved in d1/, d2/ (two working directory of datanode)

Programming Assignment 3

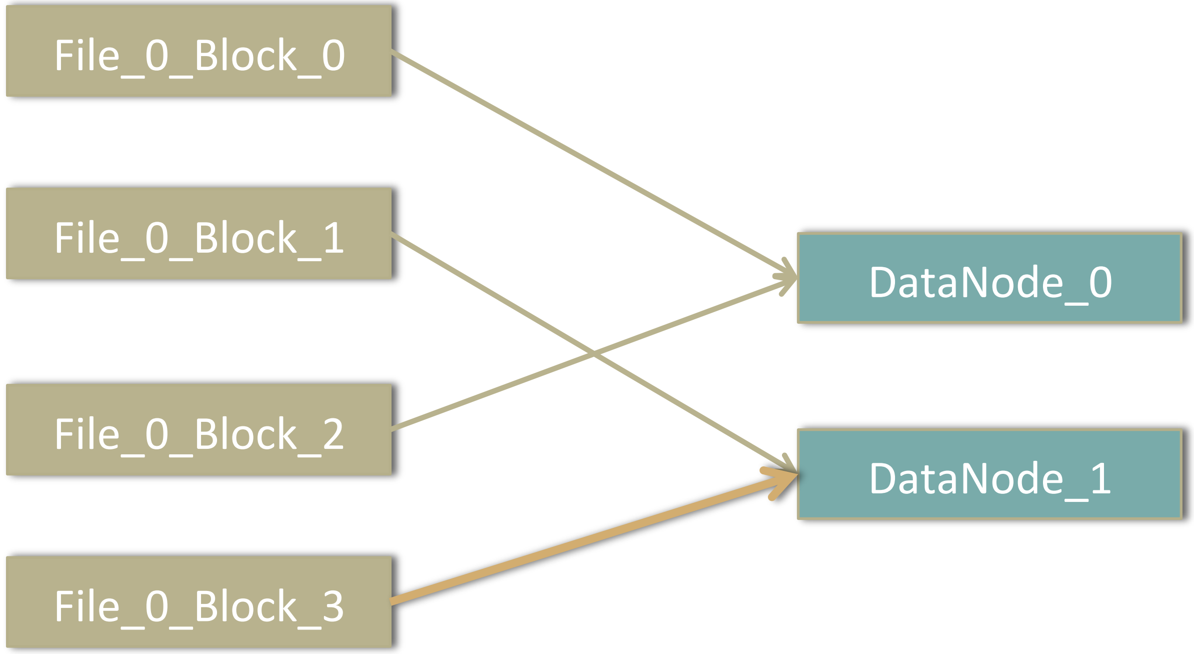
- The pairs of Test case 1&2, 3&4, 5&6 are similar
  - 2, 4, 6
    - In test.c, the file created in previous test is backup and the original copy is deleted
    - In test_2, 4, 6, send_file_request is called to send a read request is to the namenode
    - dfs_client.c, pull_file is called to send the request via socket
    - dfs_namenode.c, mainLoop(), accepts requests from client and calls the handler function (the handler functions are given)
    - dfs_namnode.c, int get_file_location(int client_socket, dfs_cm_client_req_t request), sent location information of all datanodes storing the file via socket to the client
    - dfs_client.c, in pull_file(), client receives location information and send read request to the datanodes,
    - dfs_datanode.c, mainloop(), accept request from client and calls the read_block() to return data to the client
    - test.c, compare the data returned back from datanode and the backup of the original file

Programming Assignment 3

- Test 7
  - You have to read back three files
  - Test your implementation in metadata structure (dfs_namenode.c)
    - dfs_cm_file_t* file_images[MAX_FILE_COUNT];
  - dfs_cm_file_t
    - typedef struct _file_descriptor_
    - {
    -                 char filename[256];//file name
    -                 dfs_cm_block_t block_list[MAX_FILE_BLK_COUNT];//map the block number to the datanode location
    -                 int file_size;//file size
    -                 int blocknum;//total number of blocks (actually chunks) in this file
    - }dfs_cm_file_t;

Programming Assignment 3

- Test 8 (append to a file)



Programming Assignment 3

- Test 8
  - In test.c, append_data, append 1024 bytes in "local_file"
  - In dfs_client.c, int modify_file(char *ip, int port, const char* filename, int file_size, int start_addr, int end_addr) is called
    - a modify request is sent to namenode
  - In dfs_namenode.c, int get_file_update_point(int client_socket, dfs_cm_client_req_t request)
    - Return location information of all datanodes which are used to store the data to client via socket
  - In dfs_client.c, int modify_file(char *ip, int port, const char* filename, int file_size, int start_addr, int end_addr)
    - Client send the updated data to datanodes according to start_addr, and end_addr

Programming Assignment 3

- Test 9, 10
  - Merge with your PA2
  - If you passed test cases in PA2, it does not mean that you can pass these two without any debugging process
    - Bug is always there, just depends on whether you test against the certain function point
- Watching out for partial sends / recvs
  - Data is transmitted via packets
  - Read/write, send/recv functions always return when it finishes the recving of a single packet
    - E.g. each packet is 1400 bytes, while our file size is 2800 bytes, a simple read() calling will return when it finishes reading the first 1400 bytes causing the next 1400 bytes lost
  - The most tricky part in this assignment

FAQ

- I maintained a list of FAQ in discussion board
- How to start your namenode, datanode,
  - See test.sh
  - You'd better start the programs manually for easy debugging
- "port has been used?"
  - We assume the server will run infinitely, so I didn't close the socket explicitly
    - ps aux | grep 'namenode' (or datanode)
    - Get the PID of the process
    - Kill it
    - Wait for a while (Trottier machine usually has a long latency to close the socket)
    - Restart the program

Thank You !
Q & A