Laboratory work #2

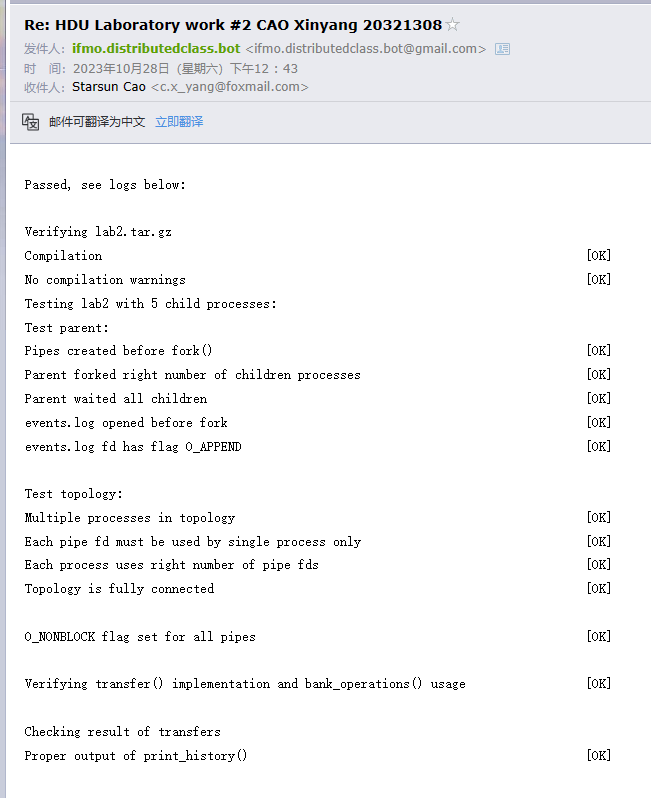
Distributed banking system

Student: CAO Xinyang  
Student HDU ID: 20321308

Mail:

c.x\_yang@foxmail.com

Screenshot or copy with mail with “Passed”:



Control questions:

1. Transfer request. Is it synchronous or asynchronous in our model from point of client? Why?  
What is difference between synchronous and asynchronous operations?

Answer:

Asynchronous. Clients kick off transfers and carry on with their tasks without having to wait for the transfers to finish. Unlike synchronous operations that hold up clients until they get immediate results, asynchronous operations let clients move forward and receive results at a later time. This method boosts both the efficiency and responsiveness of the system in a distributed setting.

2. What will happened, if transfer operations isn’t instantaneous?  
Can we calculate total amount of money at each moment of time? Why?

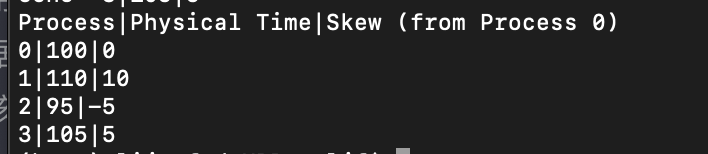
Answer:

When transfer operations isn’t instantaneous, determining the total sum of money at any given point becomes a complex task. The presence of unfinished transfers and the variability in execution times can lead to discrepancies in account balances. The sequence in which transfers take place affects the state of the system, adding complexity to the real-time balance calculation. The non-immediate nature of transfers keeps the system in a state of continuous change, making it impractical to compute an exact total balance.

3. Make an experiment. You can find in header banking.h function get\_physical\_time\_skew(). Use it instead of get\_physical\_time(). This function provides non-synchronized clocks. Why result of execution is different?  
Answer should contain output table.

Answer:

Replacing get\_physical\_time() with get\_physical\_time\_skew() results in unsynchronized clocks, leading to variations in the execution outcomes. Each process now operates its own clock with different time skews, causing inconsistencies in event timestamps. This impacts the sequence of event handling and message ordering, potentially leading to unforeseen behavior and coordination problems in distributed systems. The lack of synchronized clocks underscores the difficulties in preserving event sequencing and synchronization in such settings.



Source code:

#include <stdio.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include "labs\_headers/message.h"

#include "labs\_headers/log.h"

#include "labs\_headers/process.h"

#include "labs\_headers/banking.h"

void update\_history(BalanceHistory \*history, timestamp\_t time, balance\_t amount)

{

    int last\_time = history->s\_history[history->s\_history\_len - 1].s\_time;

    int last\_balance = history->s\_history[history->s\_history\_len - 1].s\_balance;

    for (int i = last\_time + 1; i < time; i++)

    {

        history->s\_history[history->s\_history\_len].s\_time = i;

        history->s\_history[history->s\_history\_len].s\_balance = last\_balance;

        history->s\_history[history->s\_history\_len].s\_balance\_pending\_in = 0;

        history->s\_history\_len++;

    }

    history->s\_history[history->s\_history\_len].s\_balance = amount;

    history->s\_history[history->s\_history\_len].s\_time = time;

    history->s\_history[history->s\_history\_len].s\_balance\_pending\_in = 0;

    history->s\_history\_len++;

}

balance\_t now\_balance(BalanceHistory \*history)

{

    return history->s\_history[history->s\_history\_len - 1].s\_balance;

}

void parent\_work(int count\_nodes)

{

    AllHistory all\_history;

    all\_history.s\_history\_len = count\_nodes - 1;

    /\* STUDENT IMPLEMENTATION STARTED \*/

    /\* Implement starting synchronization \*/

    for (int i = 1; i < count\_nodes; i++)

    {

        Message msg;

        receive(i, &msg);

        if (msg.s\_header.s\_magic == MESSAGE\_MAGIC && msg.s\_header.s\_type == STARTED)

        {

            continue;

        }

    }

    /\* Useful work \*/

    bank\_operations(count\_nodes - 1);

    /\* Implement finishing synchronization and collecting AllHistory \*/

    Message msg;

    fill\_message(&msg, STOP, get\_physical\_time(), NULL, 0);

    send\_multicast(&msg);

    for (int i = 1; i < count\_nodes; i++)

    {

        Message msg;

        receive(i, &msg);

        if (msg.s\_header.s\_magic == MESSAGE\_MAGIC && msg.s\_header.s\_type == DONE)

        {

            continue;

        }

    }

    for (int i = 1; i < count\_nodes; i++)

    {

        Message msg;

        receive(i, &msg);

        if (msg.s\_header.s\_magic == MESSAGE\_MAGIC && msg.s\_header.s\_type == BALANCE\_HISTORY)

        {

            BalanceHistory \*history = (BalanceHistory \*)msg.s\_payload;

            memcpy(&all\_history.s\_history[i - 1], history,msg.s\_header.s\_payload\_len);

        }

    }

    print\_history(&all\_history);

    return;

}

void child\_work(struct child\_arguments args)

{

    /\* Child arguments \*/

    local\_id self\_id = args.self\_id;

    int count\_nodes = args.count\_nodes;

    uint8\_t balance = args.balance;

    /\* BalanceHistory initialization \*/

    BalanceHistory history;

    history.s\_history\_len = 1;

    history.s\_id = self\_id;

    memset(history.s\_history, 0, sizeof(history.s\_history));

    history.s\_history[0].s\_balance = balance;

    for (int i = 0; i < MAX\_T; ++i) {

        history.s\_history[i].s\_time = i;

    }

    /\* System process identifiers used for logs \*/

    pid\_t self\_pid = getpid();

    pid\_t parent\_pid = getppid();

    /\* STUDENT IMPLEMENTATION STARTED \*/

    char buf[BUF\_SIZE];

    Message msg;

    timestamp\_t current = get\_physical\_time();

    snprintf(buf, BUF\_SIZE, log\_started\_fmt, current, self\_id, self\_pid, parent\_pid, balance);

    fill\_message(&msg, STARTED, current, buf, strlen(buf));

    send\_multicast(&msg);

    shared\_logger(buf);

    Message recv\_msg;

    int cnt = 0;

    for (int i = 1; i < count\_nodes; i++) {

        if (i == self\_id)

            continue;

        receive(i, &recv\_msg);

        if (recv\_msg.s\_header.s\_magic == MESSAGE\_MAGIC && recv\_msg.s\_header.s\_type == STARTED) {

            cnt++;

        }

    }

    if (cnt == count\_nodes - 2) {

        current = get\_physical\_time();

        snprintf(buf, BUF\_SIZE, log\_received\_all\_started\_fmt, current, self\_id);

        shared\_logger(buf);

    }

    cnt = 0;

    while (true) {

        Message req\_msg;

        receive\_any(&req\_msg);

        if (req\_msg.s\_header.s\_magic == MESSAGE\_MAGIC && req\_msg.s\_header.s\_type == TRANSFER)

        {

            TransferOrder \*order = (TransferOrder \*)req\_msg.s\_payload;

            if (order->s\_src == self\_id) {

                current = get\_physical\_time();

                update\_history(&history, current, now\_balance(&history) - order->s\_amount);

                snprintf(buf, BUF\_SIZE, log\_transfer\_out\_fmt, current, self\_id, order->s\_amount, order->s\_dst);

                shared\_logger(buf);

                Message response\_msg;

                fill\_message(&response\_msg, TRANSFER, current, order, strlen((const char\*)order));

                send(order->s\_dst, &response\_msg);

            } else if (order->s\_dst == self\_id) {

                current = get\_physical\_time();

                update\_history(&history, current, now\_balance(&history) + order->s\_amount);

                snprintf(buf, BUF\_SIZE, log\_transfer\_in\_fmt, current, self\_id, order->s\_amount, order->s\_src);

                shared\_logger(buf);

                Message response\_msg;

                fill\_message(&response\_msg, ACK, current, NULL, 0);

                send(0, &response\_msg);

            }

        } else if (req\_msg.s\_header.s\_magic == MESSAGE\_MAGIC && req\_msg.s\_header.s\_type == STOP) {

            current = get\_physical\_time();

            snprintf(buf, BUF\_SIZE, log\_done\_fmt, current, self\_id, now\_balance(&history));

            shared\_logger(buf);

            Message response\_msg;

            fill\_message(&response\_msg, DONE, current, buf, strlen(buf));

            send\_multicast(&response\_msg);

            break;

        } else if (req\_msg.s\_header.s\_magic == MESSAGE\_MAGIC && req\_msg.s\_header.s\_type == DONE) {

            cnt++;

        }

    }

    while (cnt != count\_nodes - 2)

    {

        Message msg;

        receive\_any(&msg);

        if (msg.s\_header.s\_magic == MESSAGE\_MAGIC && msg.s\_header.s\_type == DONE)

        {

            cnt++;

        }

    }

    if (cnt == count\_nodes - 2)

    {

        current = get\_physical\_time();

        snprintf(buf, BUF\_SIZE, log\_received\_all\_done\_fmt, current, self\_id);

        shared\_logger(buf);

    }

    Message history\_msg;

    current = get\_physical\_time();

    fill\_message(&history\_msg, BALANCE\_HISTORY, current, &history, sizeof(history));

    send(0, &history\_msg);

}

void transfer(local\_id src, local\_id dst,

              balance\_t amount)

{

    TransferOrder order = {src, dst, amount};

    /\* STUDENT IMPLEMENTATION STARTED \*/

    Message msg;

    fill\_message(&msg, TRANSFER, get\_physical\_time(), &order, sizeof(order));

    send(src, &msg);

    receive(dst, &msg);

    if (msg.s\_header.s\_magic == MESSAGE\_MAGIC && msg.s\_header.s\_type == ACK)

    {

        return;

    }

}

/\* STUDENTS SHOULD NOT CHANGE THIS FUNCTION \*/

\_\_attribute\_\_((weak)) void bank\_operations(local\_id max\_id)

{

    for (int i = 1; i < max\_id; ++i) {

        transfer(i, i + 1, i);

    }

    if (max\_id > 1) {

        transfer(max\_id, 1, 1);

    }

}