Laboratory work #3

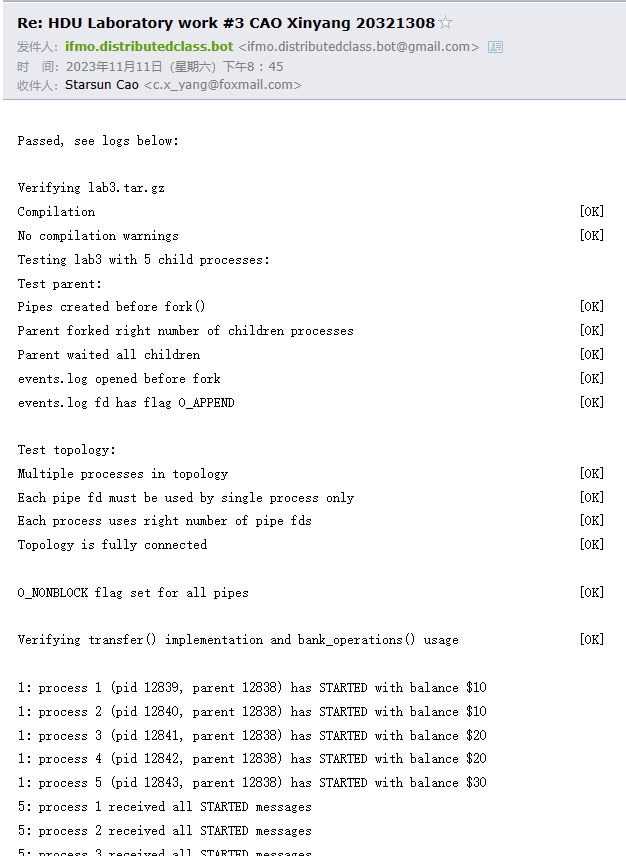
Lamport’s logical clocks

Student: CAO Xinyang  
Student HDU ID: 20321308

Mail:

[c.x\_yang@foxmail.com](mailto:c.x_yang@foxmail.com)

Screenshot or copy with mail with “Passed”:



Control questions:

1. Lamport’s clocks. What does it mean, that these clocks aren’t strongly consistent?

Answer:

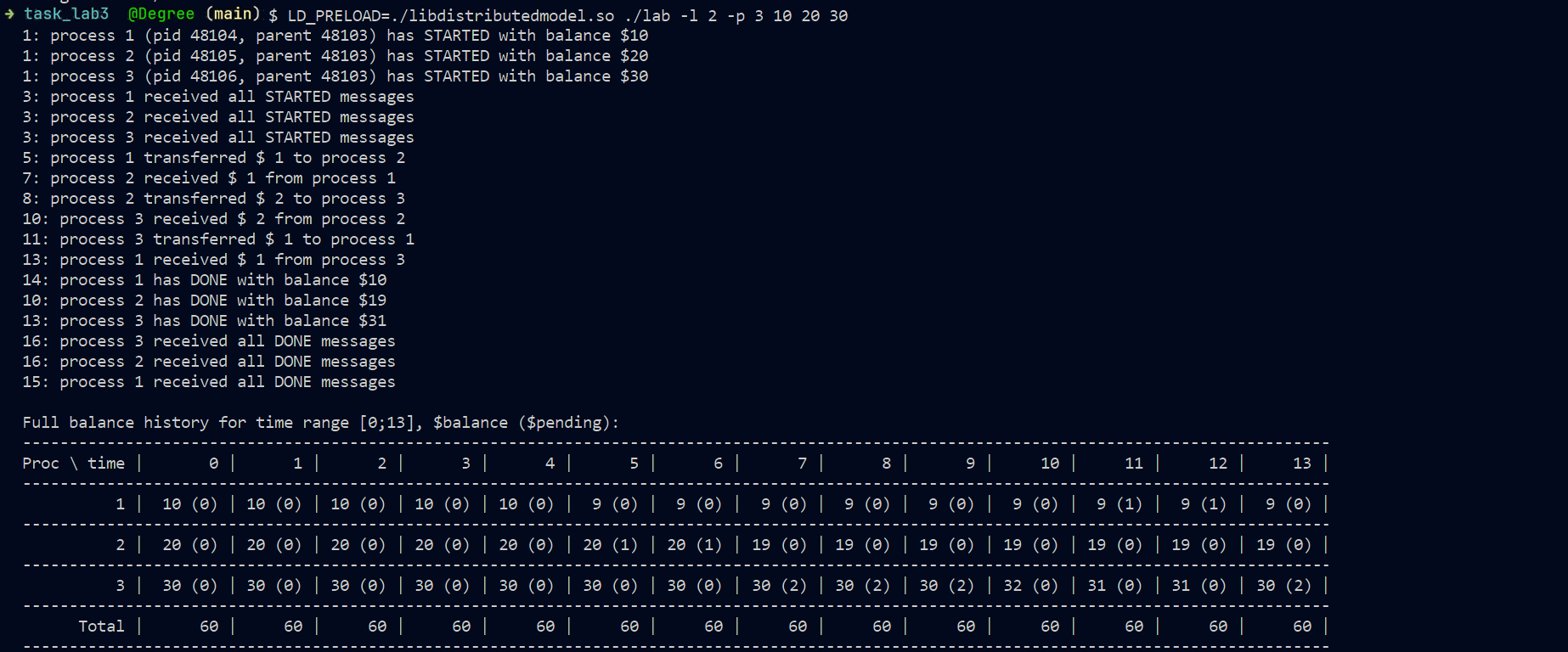
Lamport’s clocks, also known as logical clocks, are a mechanism used in distributed systems to order events. They provide a way to generate timestamps for events that reflect their causal ordering, which is a crucial aspect in understanding the behavior of distributed systems.

However, Lamport’s clocks are not strongly consistent. This means that while they can guarantee that if an event a causally precedes an event b, then the timestamp of a will be less than the timestamp of b, the converse is not necessarily true. In other words, if the timestamp of an event a is less than the timestamp of an event b, it does not guarantee that a causally precedes b.

2. What will happened, if parent process’s messages (e.g., TRANSFERs from parent) aren’t marked with Lamport’s clock timestamp?   
Modify your model and explain results. Attach resulted table to the report.

Answer:

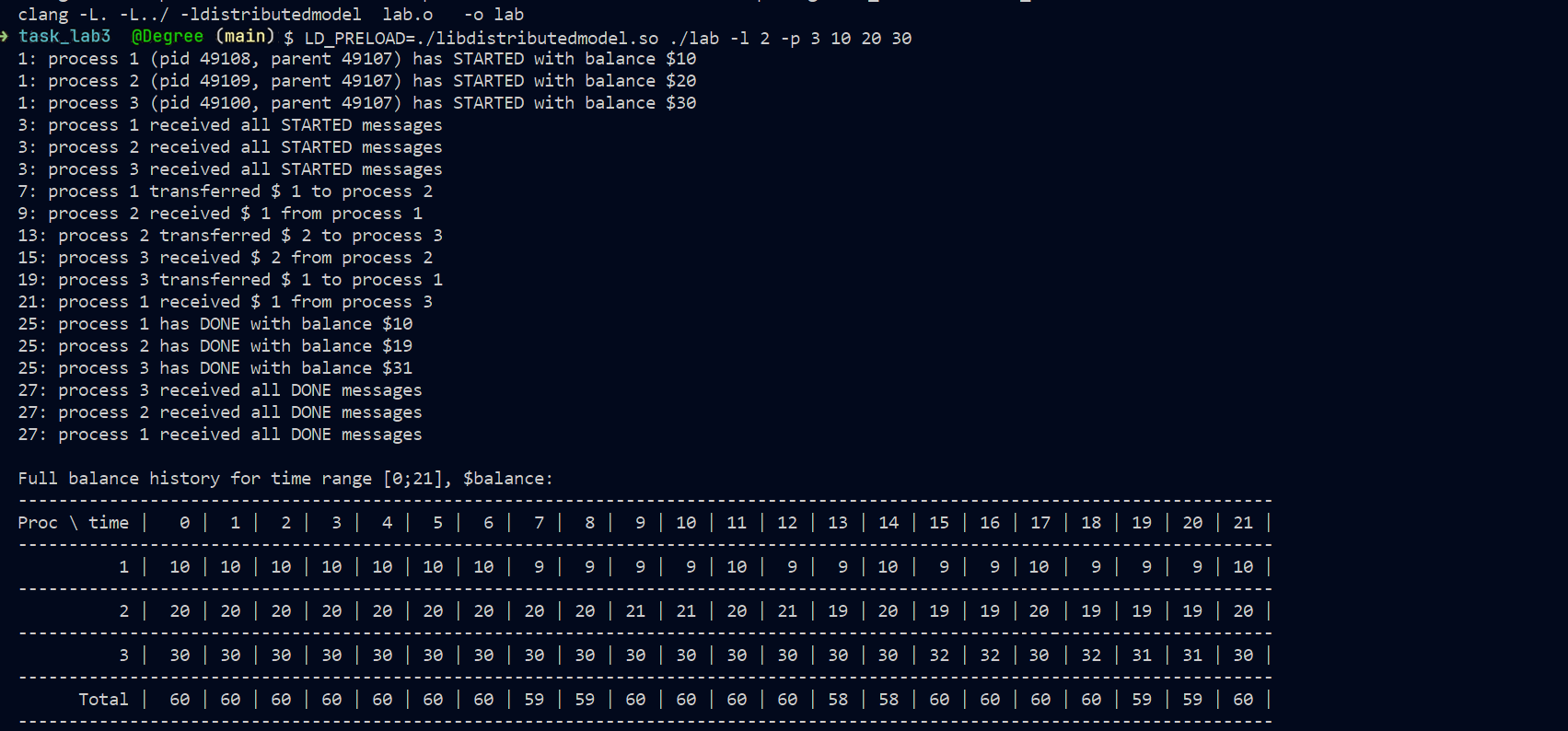
The termination of the child process is governed by its own timeline, unaffected by the parent’s schedule.



3. Make an experiment. What will happened if we will not use s\_balance\_pending\_field?  
Modify your model and explain results. Attach resulted table to the report.

Answer:

If the “s\_balance\_pending\_field” is not included in a specific model or system, it may lead to an unreported amount during the communication within a channel, likely due to delays.



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"

timestamp\_t Lamport\_time = 0;

int max(int a, int b) {

return (a > b) ? a : b;

}

void update\_history(BalanceHistory \*history, timestamp\_t pending\_start\_time, timestamp\_t pending\_end\_time, balance\_t amount, balance\_t pending\_money)

{

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;

int i = last\_time + 1;

while (i < pending\_start\_time) {

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++;

i++;

}

i = pending\_start\_time;

while (i < pending\_end\_time) {

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 = pending\_money;

history->s\_history\_len++;

i++;

}

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

history->s\_history[history->s\_history\_len].s\_time = pending\_end\_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);

Lamport\_time = max(Lamport\_time, msg.s\_header.s\_local\_time) + 1;

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, ++Lamport\_time, NULL, 0);

send\_multicast(&msg);

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

{

Message msg;

receive(i, &msg);

Lamport\_time = max(Lamport\_time, msg.s\_header.s\_local\_time) + 1;

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);

Lamport\_time = max(Lamport\_time, msg.s\_header.s\_local\_time) + 1;

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 = ++Lamport\_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);

Lamport\_time = max(Lamport\_time, recv\_msg.s\_header.s\_local\_time) + 1;

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

cnt++;

}

}

if (cnt == count\_nodes - 2) {

current = Lamport\_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);

Lamport\_time = max(Lamport\_time, req\_msg.s\_header.s\_local\_time) + 1;

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 = ++Lamport\_time;

update\_history(&history, current, current, now\_balance(&history) - order->s\_amount, 0);

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 = max(Lamport\_time, req\_msg.s\_header.s\_local\_time) + 1;

update\_history(&history, req\_msg.s\_header.s\_local\_time, current, now\_balance(&history) + order->s\_amount, 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 = max(Lamport\_time, req\_msg.s\_header.s\_local\_time) + 1;

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);

Lamport\_time = max(Lamport\_time, msg.s\_header.s\_local\_time) + 1;

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

{

cnt++;

}

}

if (cnt == count\_nodes - 2)

{

current = Lamport\_time;

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

shared\_logger(buf);

}

Message history\_msg;

current = ++Lamport\_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, ++Lamport\_time, &order, sizeof(order));

send(src, &msg);

receive(dst, &msg);

Lamport\_time = max(Lamport\_time, msg.s\_header.s\_local\_time) + 1;

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)

{

int i = 1;

while (i < max\_id) {

transfer(i, i + 1, i);

i++;

}

if (max\_id > 1) {

transfer(max\_id, 1, 1);

}

}