**Inspiration for the project:**

<https://www.youtube.com/watch?v=RY_2gElt3SA&t=4s>

**Background**: Let us think about an example of updating two different types of databases. The first example involves an online ticket purchase at Ticketmaster. In this case, each ticket is unique, with one person corresponding to one seat.

Ticketmaster comprises not just one computer but many distributed computers. The computer used by a person in Baltimore used to purchase a ticket is likely different from the computer accessed by someone in Seattle. Although they may appear as a single entity, these distributed computers must work together to ensure that the same ticket is not sold twice.

When you wish to purchase a concert ticket, you first use your mobile device or your browser to access the Ticketmaster website. At the website, you determine which seat location you can afford. You choose your seat and enter your payment information. At this point, the seats you wish to purchase will be locked in the Ticketmaster database for a short time until you finish your transaction. If the transaction completes, the tickets are removed from the list of available tickets. If you do not finish in time, the lock will be released and put back in the list of available tickets for another person to purchase the seat. No one else can buy those seats until the time expires or the seats are purchased.

On to the second example. What is the difference between reserving a specific seat at a concert and calculating the number of likes on YouTube? With the number of likes, we are not reserving one specific seat but incrementing the “likes” counter. Each “like” on YouTube is just as valuable as another. We are not purchasing anything unique. It is more like voting for a candidate. With a huge website such as Meta and YouTube, there are many computers and many people clicking like on these computers. When we see a video with the number of likes at the bottom of the app, we see the sum of likes from different computers. Thus, the problem. Networked computers must communicate with some other server that is keeping track of the total number of likes.

**Question**: You are using YouTube and decide if you like the latest cute baby video. What happens when you press “like” on that YouTube video? How does the like count get updated? YouTube is distributed on many servers around the world and in the US. How does your “like” get recorded when there are so many YouTube servers? Does it make sense for each YouTube server to contact a central server to update each time the like button is pressed? Probably not since that would create too much network traffic. It is important to eventually record each like, but does it need to be done immediately? Not instantly, but eventually. Note: The number of likes is part of how creators get paid, so it is important to not lose a like.

How does the update process work? The remote YouTube servers will keep track of likes and update the central “like” server periodically to prevent overloading the central “like” server with every update. Because of this, the central like server is never exactly accurate at a point in time. The term for this is “[**Eventual Consistency**](https://www.youtube.com/watch?v=RY_2gElt3SA&t=4s)”.

**Student To Do**: Your task for Project 2 is to model this system. Let us keep this assignment simple and model this on a single computer and use different processes instead of different computers. Before we give you the details, you are creating three basic processes. The first process (**ParentProcess)** forks the second processes, the **LikesServer**. The third process (**PrimaryLikesServer**)collects the total number of likes from the Like Servers using the socket call.

1. You might spend some time looking at the diagram below at the end of the project description. Note each process has a log file.
2. Have a **ParentProcess** create 10 child **LikesServer**. The **ParentProcess** should be called **ParentProcess** and the **LikesServer** are the child processes and should be called **LikesServer0**,2,…,9. The parent process will create each child process(**LikesServer)** one second after the previous child. The **ParentProcess** will create a log /tmp/ParentProcessStatus. Each time the **ParentProcess** starts a child, indicate the child has been started in the log. When the child process ends, note this in the log. Include a time stamp for each entry
3. Each **LikesServer** will emulate a simplified YouTube server where the videos are displayed, and likes are generated. Each child process should live for five minutes. We will keep things very simple, and each child process will only generate a random number of likes between 0 and 42 using the rand() function. This is the number of likes that the server received during the period. Use the rand() function again to create a random number between 1 and 5 to create a random interval in seconds to notify the **PrimaryLikesServer** (to be described in part 2).

Use sockets to communicate. This is a form of “Interprocess Communication” discussed in class to transfer the information from the **LikesServer** to the **PrimaryLikesServer**. The communication must be bidirectional. The data passed to the **PrimaryLikesServer** should have the child’s process name and the count of likes. For example, “LikesServer1 14”. The **PrimaryLikesServer** should respond when it receives a count of likes from each **LikesServer**. This should be noted in a log with a file name of /tmp/LikesServer0, /tmp/LikesServer1,… After the **LikesServer** is successful and has a positive response from the **PrimaryLikesServer**, reset the running like counter in each child’s **LikesServer**.

Time to close shop and clean up. Each **LikesServer** should run for five minutes. Each **LikesServer** should notify the **ParentProcess** it is finished by exiting with a zero-return code if there were no errors. Indicate this in the **ParentProcess** log.

A picture containing text

Description automatically generated

1. Create a server process to keep track of the likes. Call this process the **PrimaryLikesServer**. This process will take input from the 10 **LikesServers**. To keep track of each client **LikesServer**, each **LikesServer** process will contact the **PrimaryLikesServer** using sockets. The **PrimaryLikesServer** will have a simple log of the data passed from the client **LikesServers.** See example of log below. When contacted by a **LikesServer**, the **PrimaryLikesServer** will save the information passed in the file name /tmp/PrimaryLikesLog. Hint: $ tail -f can be used to display the running output of a file to the screen. Update the “total like counter” in the log. There should be an entry in the **PrimaryLikesServer** for each transaction and an update to the running like total.
   1. Client1 15
   2. Total 15
   3. Client3 33
   4. Total 48
   5. Client2 42
   6. Total 90
2. Going back to the first part of this exercise: If the **LikesServers** process fails, notify the **ParentProcess** that forked the 10 children by exiting the **LikesServers** process with a “1” return code. The **ParentProcess** should indicate this in the parent log. The parent process should keep track of the children **LikesServers** when they start and finish by keeping a log file. Note: Log files are great for debugging.

**Hints and thoughts about the program:**

1. Let us think about secure programming for a minute. Data is being passed from the **LikesServers** to the **PrimaryLikesServer**. The first field will be something like the word “child” followed by an integer. The second field will be an integer between 0 and 42. An example is “Child4 15”. Make sure this is true by testing for this. Test for invalid data and do not accept it.
2. You might be asking yourself how many times each file should be opened and closed. Unless the file is closed each time there is an update, there is a possibility of lost data if the server crashes. Thus, open and close the file every time there is an update.
3. In the real world, these servers will be on many computers. IPC must account for the data transfer. Thus, techniques such as shared memory, message passing, and pipes will not work in this case. You are required to use sockets.
4. How should you start solving the project? Break it down into parts and solve each one before continuing. Write down each step. Do not write the entire program and attempt to figure out a problem. One step at a time works best. Keep it simple.
   1. Write a program, **ParentProcess** that forks/exec ten times and creates ten child processes, **LikesServer0**-9. It should originally be a printf(“Hello, world”); program. Keep it simple.
   2. Put a printf in the **PrimaryLikesServer** and each **LikesServer** indicating it is alive. Use this for debugging and take it out when the project is working.
   3. Save to GitHub!!!
   4. Write a simple test program that uses sockets and pass data between the programs. Find example code on the web, Bard, or ChatGPT. Make sure you understand all parameters you use in case I ask you about this on a test. Get this working before proceeding.
   5. The final process to create is the **PrimaryLikesServer** . It will accept IPC connections from the **LikesServer0**-9. This is a server process and does not exit.
   6. Send a positive return code to the parent process and a failure code to the parent. This is actually exit() system call.
   7. Next, add a random number to the **LikesServer**.
   8. Add your Interprocess Communication code to the **LikesServer** and **PrimaryLikesServer**. This is where you use the internet for model code. I strongly suggest you write a simple toy program before integrating this into your project. Once you get it working, save it to GitHub.
   9. Once the communication is working between the **LikesServer** and the **PrimaryLikesServer,** add logic for Eventual Consistency.
   10. Check for bad data in **PrimaryLikesServer**. If you receive bad data, note it in the log and do not process it.
   11. Put the appropriate logging in the process **LikesServer** and **PrimaryLikesServer**.
   12. Wait for five minutes and let the **LikesServer** complete.
   13. Then, do the close-up work.
5. Conditional assembly is a good way to develop a project. #define Debug 1 is a macro you can use to check as your code is compiled. In your code, you can check for this variable if you use #ifdef Debug. Appropriate uses for this are debugging printf statements that can easily be removed by #define Debug 0. Use the following for a longer description.

<https://www.educative.io/answers/what-is-the--sharpifdef-directive-in-c>

1. Extra credit: This is an extremely insecure implementation. Implement authentication using the openssl library. No hints given here.
2. Extra credit: You can earn on point per day if you turn the project in early up to five points on this project.
3. This project is about forking and Interprocess communication and not about kernel programming.
4. Put information on how to turn the project in. (Sanaa does this) The grader should be able to copy all files in this directory and compile the project. Post your code to GitHub.
5. You should have 12 processes running concurrently.
   1. **ParentProcess**(1)
   2. **LikesServer**(10)
   3. **PrimaryLikesServer**(1)
6. **ParentProcess** starts the 10 **LikesServers**. The **PrimaryLikesServer** should be started separately. The **PrimaryLikesServer** does not exit.
7. If you are going to be late, you must notify your grader.
8. Put the date you wish the grader to pull from GitHub in your README.
9. Each process has a log. Look below for an example of what they should look like. The grader will look at these logs to determine if your program is working correctly.

**10 LikesServer**s

**PrimaryLikesServer**

**ParentProcess** 

Child0 Started

…

Child4 ended

LikesServer1 14

Total 14

LikesServer2 10

Total 24

…

LikesServer 14 0

…

Fork / exec

Socket()

Note: 14 is the number of likes in the period. 0 is the successful return code from the IPS to the PrimaryLikesServer

The LikesServers send an exit code to the ParentProcess. The ParentProcess logs this code.