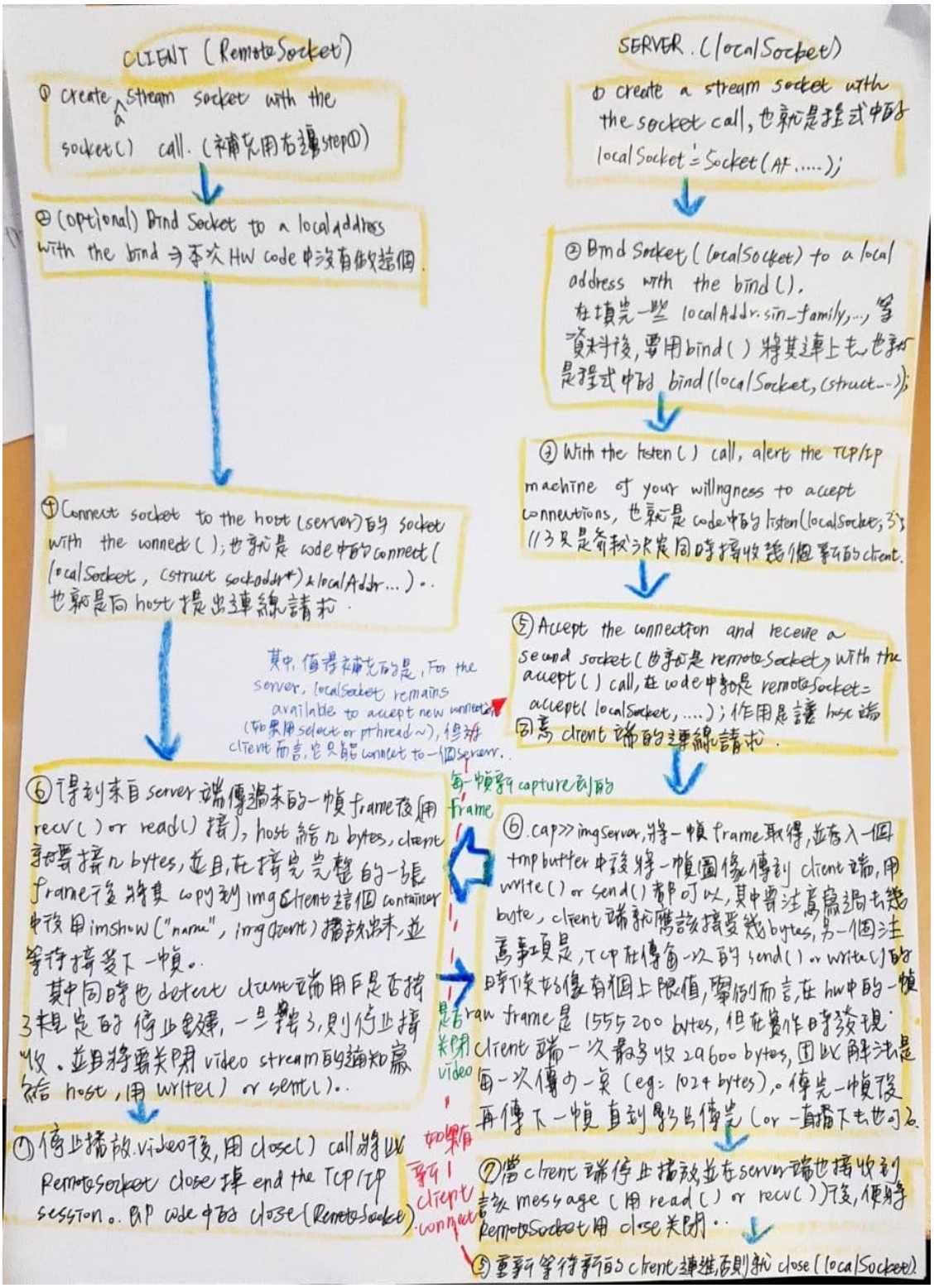
CN HW2

B05611047 鍾宜樺

1. Draw a flowchart of the video streaming and explains how it works in detail. (5%)

下面的 flowchart 是 server 端沒有 select、client 端也沒有做 buffering，就是讀一幀 frame 就 imshow 一幀 frame 的 flowchart，並且假設我們想要播放的影片對方是擁有的繪製下圖。

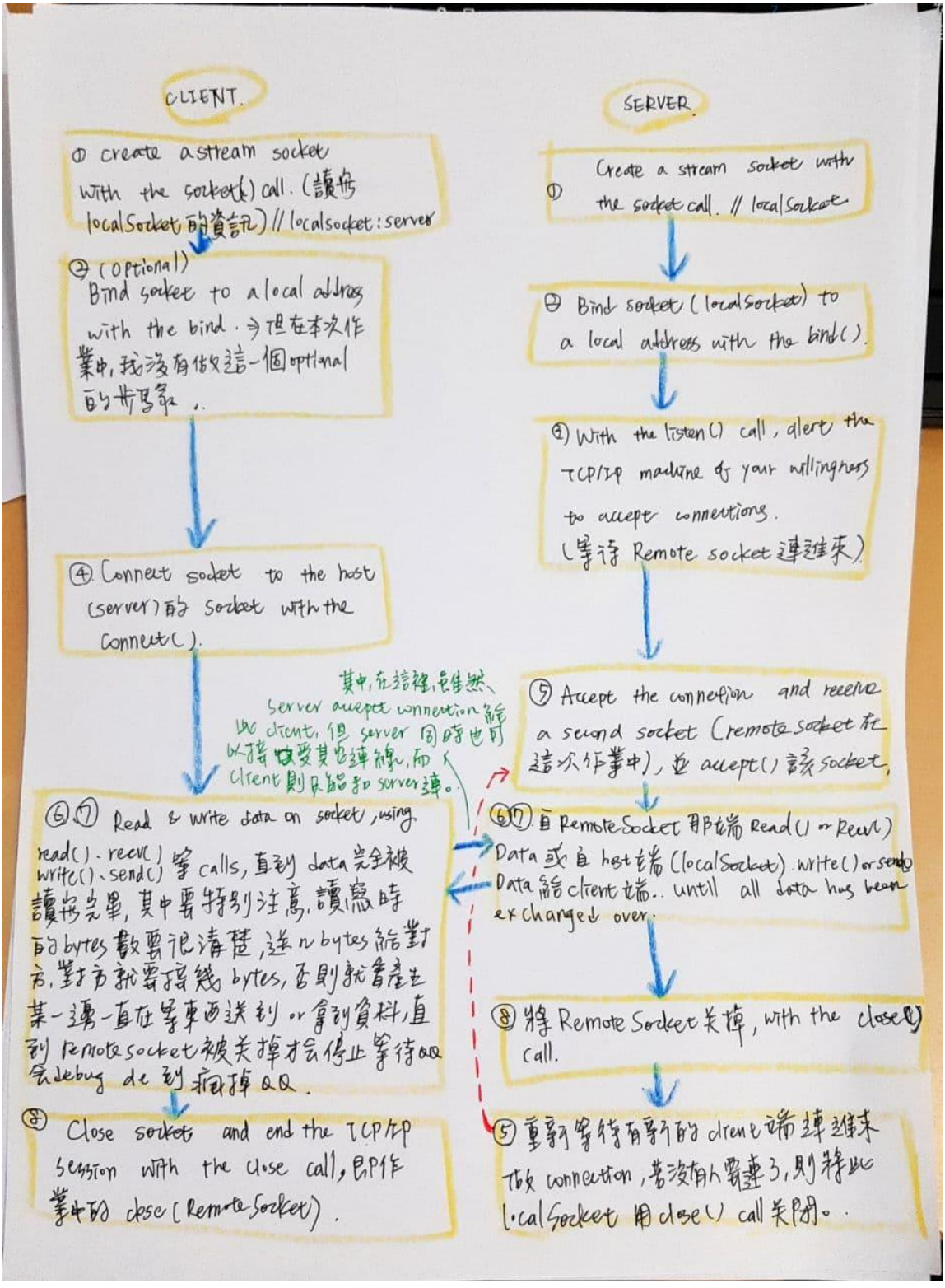
其中想要再補充一點是，下圖中提到的可以確認傳送幾 bytes 就要確定對方接收幾 bytes 的原因是因為，他你用 write 或 send 的時候他都會有一個 return 值告訴你他實際上傳出了多少 bytes。而另一端在接收的時候也會有一個 return 值可以知道他接收了幾 bytes。 Debug 的時候可以用這兩個值確認你的東西有沒有傳對，如果值不一樣，有可能是你前面的 read 還沒有結束所以後面的訊息被吃掉了，或者是有什麼更悲慘的 bug 。總之這些 return 值很好用。



2. Draw a flowchart of the file transferring and explains how it works in detail. (5%)

其中如果上面那張圖有敘述到的地方可能就不會在多加贅述一次。為了精簡圖畫，下圖是假設我們想要的檔案對方是擁有的而繪製的。如果對方沒有該檔案，則就可以直接關掉此 remoteSocket 重新連線開啟新一輪的 connection。

在 file transfer 的時候，先確認你想要的檔案在對方的所有檔案中是否存在，假設我們是 client 端，對方式 server。我想從 server 端要一個檔案過來，我要先將我想要的檔案名稱傳給 server 端詢問該檔案是否存在後(用 write() or send())。假若該檔案存在那才會開始 file transfer。確認檔案存在之後就可以開始進行傳資料的動作，傳資料則就是 send() write()都可以。並且在本地端要開一個新的 file 去接收後 server 端傳來的資料後，在本地端寫入該新的檔案中(用 fopen, fwrite 就可以了)一直到接收完資料並寫入完資料後，利用 read() recv(0 的 return 值確認資料接收完便可將 remoteSocket 關閉。



3. What is SIGPIPE? It is possible to happen to your code? If so, how do you handle it? (5%)

(1) 從網路上的資料看來，這好像是 TCP 協定容易出現的問題。白話文是說：

用協議 TCP 的 socket 程式設計，host 端沒辦法知道 client 已經關閉了 socket，因為不知道 client 已經關閉 socket，導致 host 端還是不斷在向該已關閉的 remoteSocket 上 send 資料的時候，就會導致 SIGPIPE 發生。如果嘗試 send 到一個已關閉的 socket 上兩次，就會出現 SIGPIPE 的訊號。

而系統預設產生 SIGPIPE 訊號的措施是關閉程序，也就是說遇到這個狀況的時候 server 端傳送兩次訊號都沒有 client 接東西的時候 server 端也會被關閉。或者是我在寫作業的時候還會出現 core dumped 的狀況 QWQ。

接下來的分析是根據網路敘述：分析 TCP 協議的缺陷以至於伺服器無法及時判斷對方 socket 已關閉：具體的分析可以結合 TCP 的"四次握手"關閉。TCP 是全雙工的通道,，可以看作兩條單工通道，TCP 連線兩端 (也就是作業中的 localSocket 以及 remoteSocket) 兩者各負責一條。假設 client 端呼叫 close 時，雖然本意是關閉整個兩條通道，但 host 端只是收到 FIN 包，按照 TCP 協議的語義，表示 client 端只是關閉了自己負責的那條單工通道，但本身仍然可以繼續接收資料。

也就是說, 因為 TCP 協議的限制，對於兩端而言，都無法獲知對端的 socket 是呼叫了 close 還是 shutdown。故會有 SIGPIPE 的 signal 出現。

(2) 在我的作業裡<u>沒有</u>發生這件事情，原因是因為一直都有在 maintain 不要讓 host 對已經關閉的 client 傳東西。

(3) 解決方法：先講我自己的做法，再來討論網路上找到的高級解決辦法 QWQ

<u>(a) 我的解法</u>：

我自己在寫作業遇到這個問題的時候，會特別在實作的時候，注意到 host (我) 傳幾 bytes 就要確認 client (對方) 也是接幾 bytes 的這個注意事項。

換句話說就是，就是基本上在每一步驟的只要有讀寫，我都會把要讀寫的 bytes 數寫死，因為不講求效能所以寧可傳垃圾過去，也不要少傳東西或者是花太多時間思考自己是不是有傳對接對。並且我在每次 host 傳過去幾 bytes 後，都要確認接收端確實有接到幾 bytes 後、數值一樣之後，我才進行下一次的 read、write。

那在作業中的 file transferring 的時候就是如同上面寫的，A 端傳給 B 端的每一步，只要有一次讀寫的動作我都會確定他們數值一樣之後，再進行下一次的讀寫。並且要特別注意在資料傳輸完畢後，在 host 端和 client 端就會各自將 remoteSocket 關閉。那在 video Streaming 的部份的時候，雖然 host 端會一直將一偵一禎的 frame 傳過來給 client，但此時 client 端視開著地所以不會發生 SIGPIPE 的問題。那一旦 client 端要關閉播放影片之後，client 端就會傳一個訊息給 host，告訴他說我要關閉了，然後 host 一旦接收到那個 client 要關閉播放影片功能訊號後變會停止傳送東西給 client 並且關閉這次的 remoteSocket 的 connection。因此大概維護這些事情在作業中就不會出現 SIGPIPE 的狀況。

<u>(b) 接著是網路上看到的大神做法：</u>

重新定義遇到 SIGPIPE 的措施，signal(SIGPIPE, SIG_IGN)；具體措施在函式 SIG_IGN 裡面寫。為了避免程序退出，可以捕獲 SIGPIPE 訊號，或者忽略它，給它設定 SIG_IGN 訊號處理函式：signal(SIGPIPE, SIG_IGN)；這樣,第二次呼叫 write 方法時他的 return 值會是-1。同時 errno 置為 SIGPIPE。如此一來，其中一端便可以能知道對端已經關閉。

<u>長篇大論的整理 (摘自 Ref 2-1)，因為感覺寫出上面的做法這題應該就可以了，因此將這段很長的補充放在作業的最後面。如果這裡寫得不夠多的話，麻煩助教看到最後補充(1)QQ 感謝</u>

4. Is blocking I/O equal to synchronized I/O? Please give me some examples to explain it. (5%)

(1) blocking I/O equal to synchronized I/O 是否一樣：

不一樣。其中 blocking I/O 被包含在 synchronized I/O 中。

根據 Richard Stevens' **"UNIX? Network Programming Volume 1, Third Edition: The Sockets Networking", section 6.2 "I/O Models"**的教科書中摘錄定義如下：

*** blocking I/O:** the characteristic of blocking IO is that it is blocked in two stages of IO execution (two stages of waiting for data and copying data).

*** synchronized I/O:** A synchronous I/O operation causes the requesting process to be blocked until that I/O operation completes;

根據上述定義我們就可以發現 blocking I/O 被包含在 synchronized I/O 中。因為 blocking I/O 在給 system kernel 要求後，變會一直等待，直到 system kernel 把東西準備好給予他回復，他都不會去做其他的事情，都是在原地等待。等到一件事情完成之後才能去做下一件事情，符合上面 synchronized I/O 的定義。Process 會被 block 住，一直到這個動作完成才可以進行下一個 operation。

但 synchronized I/O 不一定為 blocking IO。因為 synchronized I/O 有可能是別種狀況例如 non-blocking IO，接著敘述為什麼 non-blocking IO 被包含在 synchronized I/O：

介紹 non-blocking IO：In non-blocking IO, the user process actually needs to constantly actively ask the kernel data is ready. When the IO request is added with a flag bit such as O_NONBLOCK, it returns immediately. If the IO is not ready, it will return an error. The request process needs to actively poll to continuously send IO requests until it returns correctly.

也就是說 non-blocking IO 會不斷去問 system kernel 那邊準備好了沒有，如果還沒他就會回傳 error，並且繼續問下去一直到 system kernel 回傳說東西準備好了。那換句話說我們也會發現，其實在 non-blocking IO 他雖然會一直問，但他同樣 process 除了詢問的動作之外，也不能進行其他的動作，也就是說 process 也會被 block 在這個 operation，直到這個 system kernel 回覆說東西準備好了他才可以進行下一個 operation。因此 non-blocking IO 也是 synchronized I/O 的一種。

但 non-blocking IO 和 blocking I/O 是不一樣的，如前所述可知，blocking I/O 會讓 user 的 process block 住，直到操作完成才會繼續作業。而 non-blocking 則是在 kernel 還在準備資料的情況下會立刻回傳 (直到東西準備好前，會一直 repeat 詢問、回傳 error)。

因此可知 blocking I/O 被包含在 synchronized I/O，但反之不然。

(2) 舉例說明：

其中，因為 synchronized I/O 包含很多種 IO 狀況，分別有：blocking IO, non-blocking IO, IO multiplexing, signal-driven IO。那在本題中只舉例子去敘述，不多加對各種 IO 做贅述。

舉例今天大家去一家餐廳吃飯，每個人都想要吃炒飯，那就會出現各種不同的等待餐廳把飯做好的情形：

(a) blocking IO：這人非常乖巧，他要求一盤炒飯，則他就只會訂購一盤炒飯。點餐完後就坐在椅子上等待，並且在等餐廳把飯炒好的過程中，他什麼也不做。一直到服務員說開始用餐並將炒飯放到他桌上他才會開始下一個動作，吃飯。

(b) non-blocking IO：這個人也是一樣的，只要求一盤炒飯就只訂購一盤炒飯，然後開始等待。但這個人很煩，他每過一陣子就會問服務員說：炒飯準備好了嗎？，如果還沒他就會臉很臭(回傳 error)然後繼續等待。等到再過了一會兒，再跑去問…，不斷重複這個過程，直到服務員說：飯炒好了，並且服務員把炒飯放到他桌上，他才會再開始下一個和

等待炒飯時不同的動作，也就是開始用餐。

接下來下面的(c)、(d) 在前一小題並沒有多加敘述，但我們只要知道他也是 synchronized I/O 的其中一種 IO 就好，詳情請見 Ref 中的解釋，因為題目沒有要求要介紹這兩種因此不多加贅述，但因為舉例要舉完整，還是把他們兩個帶進來了。

(c) IO multiplexing：這個也是屬於 synchronized I/O 的其中一種。這人就比較特別了，他會同時 order 很多盤炒飯，然後開始等待。在等待的時候他甚麼也不會做。那為什麼要點很多盤炒飯的原因是因為這個人假設：在某個時候，廚房可以同時準備一道或很多道菜，因此他一次點很多盤炒飯就有比較高的機會先被做到。他會一直等待，到服務員跑過去說：您的一些菜已經準備好了。我們現在為他們服務嗎？。這人就會回答：好的。服務員會立即上炒飯，服務員一次只能提供一盤炒飯，而他從開始吃到炒飯之後，就會一盤接著一盤的吃下去到結束。

(d) signal-driven IO：這個也是屬於 synchronized I/O 的其中一種。他只訂購一盤炒飯，然後將手機號碼留給服務員後他就跑出去玩了，告訴他一旦炒飯炒好了可以立即打電話給他了，但同時服務員還不會將炒飯端上飯桌上。這人會立即返回餐廳，對服務員說：可以立即上菜了。他就會坐在桌旁等著服務員端菜，然後吃飯。

或者是有另一個更有趣的例子，摘自 ref 3-(2)，示例來說明這上面四個 IO 模型：

有四個人 A，B，C，D 釣魚：

(a) A (blocking IO) 使用最古老的釣魚竿，因此必須將其保留，等到魚鉤上然後拉動槓桿

(b) B (non-blocking IO) 的釣魚竿具有可以顯示是否有魚鉤的功能，因此，B 會與它旁邊的 MM 聊天，過一會兒看是否有魚鉤，如果有，則迅速拉動操縱桿。

(c) C (IO multiplexing) 使用的釣魚竿與 B 使用的釣魚竿相似，但他想到了一個好方法，即同時放幾根釣魚竿，然後呆在旁邊，一旦顯示屏顯示魚被鉤住，它將拉起相應的釣魚竿。

(d) D (signal-driven IO) 是富人。富人只是僱用了一個人來幫助他釣魚，一旦該人抓到魚，他就向 D 發送了一條短信。

補充(1) 摘自 Ref 2-1,

3-(3) SIGPIPE 的解法長篇大論版本：

當伺服器 close 一個連線時，若 client 端接著發資料。根據 TCP 協議的規定會收到一個 RST 響應，client 再往這個伺服器傳送資料時，系統會發出一個 SIGPIPE 訊號給程序，告訴程序這個連線已經斷開了，不要再寫了。又或者當一個程序向某個已經收到 RST 的 socket 執行寫操作是，核心向該程序傳送一個 SIGPIPE 訊號。該訊號的預設是終止程序，因此程序必須捕獲它以免不情願的被終止。

上面這段用我的理解說就是，因為 server 端雖然接收到了 SIGPIPE 的訊號，但因為他不想要被關掉，像是這個 server 端可能同時供應很多人連線，一個人 crash 並不代表他跟其他人的連線也要毀滅，因此不希望 sever 程序被關掉的話，就要因為那個 crash 掉的 client 端得到的 SIGPIPE 訊號給攔截，才不會讓 server 一起被關掉，我在寫作業的時候就被關掉好幾次 QMQ。

接著根據訊號的預設處理規則 SIGPIPE 訊號的預設執行動作是 terminate (終止、退出)，所以另一端會退出。若不想該端退出可以把 SIGPIPE 設為 SIG_IGN 如：signal (SIGPIPE, SIG_IGN)；這時 SIGPIPE 交給了系統處理。伺服器採用了 fork 的話，要收集垃圾程序，防止殭屍程序的產生，可以這樣處理：signal(SIGCHLD,SIG_IGN)；交給系統 init 去回收。這裡子程序就不會產生殭屍程序了。

接著敘述怎麼屏蔽掉 SOGPIPE 的 signal：

在 linux 下寫 socket 的程式的時候，如果嘗試 send 到一個 disconnected socket 上，就會讓底層丟一個 SIGPIPE 訊號。這個訊號的預設處理方法是退出程序，大多數時候這都不是我們期望的。因此我們需要過載這個訊號的處理方法。

呼叫以下程式碼，即可安全的遮蔽 SIGPIPE：

```
struct sigaction sa;
sa.sa_handler = SIG_IGN;
sigaction( SIGPIPE, &sa, 0 );
```

signal 設定的訊號控制代碼只能起一次作用，訊號被捕獲一次後，訊號控制代碼就會被還原成預設值了。sigaction 設定的訊號控制代碼，可以一直有效，直到你再次改變它的設定，如下：

```
struct sigaction action;
action.sa_handler = handle_pipe;
sigemptyset(&action.sa_mask);
action.sa_flags = 0;
sigaction(SIGPIPE, &action, NULL);
void handle_pipe(int sig) {
//不做任何處理即可 }
```

而 RST 的含義為"復位"，它是 TCP 在某些錯誤情況下所發出的一種 TCP 分節。有三個條件可以產生 RST：

(1) SYN 到達某埠，但此埠上沒有正在監聽的伺服器。

(2) TCP 想取消一個已有連線。

(3) TCP 接收了一個根本不存在的連線上的分節。

1. Connect 函式返回錯誤 ECONNREFUSED: 如果對客戶的 SYN 的響應是 RST，則表明該伺服器主機在我們指定的埠上沒有程序在等待與之連線（例如伺服器程序也許沒有啟動），這稱為硬錯（hard error），客戶一接收到 RST，馬上就返回錯誤 ECONNREFUSED. TCP 為監聽套介面維護兩個佇列。

兩個佇列之和不超過 listen 函式第二個引數 backlog。 當一個客戶 SYN 到達時，若兩個佇列都是滿的，TCP 就忽略此分節，且不傳送 RST.這個因為：這種情況是暫時的，客戶 TCP 將重發 SYN，期望不久就能 在佇列中找到空閒條目。

要是 TCP 伺服器傳送了一個 RST，客戶 connect 函式將立即傳送一個錯誤，強制應用程序處理這種情況，而不是讓 TCP 正常的 重傳機制來處理。還有，客戶區別不了這兩種情況：作為 SYN 的響應，意為“此埠上沒有伺服器”的 RST 和意為“有伺服器在此埠上但其佇列滿”的 RST. Posix.1g 允許以下兩種處理方法：忽略新的 SYN，或為此 SYN 響應一個 RST.歷史上，所有源自 Berkeley 的實現都是忽略新的 SYN。

2．如果殺掉伺服器端處理客戶端的子程序，程序退出後，關閉它開啟的所有檔案描述符，此時，當伺服器 TCP 接收到來自此客戶端的資料時，由於先前開啟的那個套接字介面的程序已終止，所以以 RST 響應。 經常遇到的問題： 如果不判斷 read , write 函式的返回值，就不知道伺服器是否響應了 RST, 此時客戶端如果向接收了 RST 的套介面進行寫操作時，核心給該程序發一個 SIGPIPE 訊號。此訊號的預設行為就是終止程序，所以，程序必須捕獲它以免不情願地被終止。 程序不論是捕獲了該訊號並從其訊號處理程式返回，還是不理會該訊號，寫操作都返回 EPIPE 錯誤。

3． 伺服器主機崩潰後重啟 如果伺服器主機與客戶端建立連線後崩潰，如果此時，客戶端向伺服器傳送資料，而伺服器已經崩潰不能響應客戶端 ACK，客戶 TCP 將持續重傳資料分節，試圖從伺服器上接收一個 ACK，如果伺服器一直崩潰客戶端會發現伺服器已經崩潰或目的地不可達，但可能需要比較長的時間； 如果伺服器在客戶端發現崩潰前重啟，伺服器的 TCP 丟失了崩潰前的所有連線資訊，所以伺服器 TCP 對接收的客戶資料分節以 RST 響應。

Ref:

1. file transferring:
https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.halc001/o4ag1.htm
2. SIGPIPE:
(1) https://www.itread01.com/content/1549922071.html
(2) https://www.itread01.com/p/184413.html
3.
(1) https://medium.com/@clu1022/%E6%B7%BA%E8%AB%87i-o-model-32da09c619e6
(2) https://titanwolf.org/Network/Articles/Article?AID=11f9621f-0577-4a17-ae59-11fdaa0cadde#gsc.tab=0
(3) https://developpaper.com/analysis-of-io-model-blocking-non-blocking-io-multiplexing-signal-driven-asynchronous-io-synchronized-io/