Socket Programming Implementing Complex TCP Servers By CSEPracticals

Networking Operating Systems Linux System Programming Kernel Network Protocols TCP/IP
Memory Management IPC RPC Multi-threading Socket Programming Asynchronous Programming

www.csepracticals.com
Telegram grp : telecsepracticals

Codes : https://github.com/sachinites/TCPServerLib

Implementing Complex TCP Servers

*

We will go beyond simple implementation of client/servers programs present all over internet (Advanced Course)

Course objective : *Learn how to implement a typical complex Networking Socket library, closely tied to thread management*

Agenda

1. How to manage Multiple Clients through Multiplexing
2. Creating Multi-Threaded Clients
3. Forcefully disconnecting the client
4. Gracefully Shutting down TCP Server
5. Notifying events to application
    1. Client new connection
    2. Client disconnection
    3. Client msg recvd
6. Detecting connection live-ness using Keep Alive msgs
7. Handling Concurrency using locks
8. TCP Msg Demarcation
9. Maintaining statistics per client connection
10. Client Migration
11. Building Socket Programming C++ Library over Posix

Pre-Requisites :

1. General Programming (any main-stream language)
2. Socket Programming Basics

    accept( ) , select( )/epoll( ) , send( ) , recv( ) , close( )
3. Basic Multi-threading (Posix pthreads)
    1. Starting a thread
    2. Thread Cancellation/Thread Join
    3. Mutexes & Semaphores
4. Timers (Starting, Restarting, Cancelling)
5. STL lists / LinkedList

Pure C Users

We will write code in C++, but we will write mostly C part of C++

- C programmers can do this course easily as well

    class → struct
    new → malloc/calloc, delete → free
    STL list → own linked lists, etc
    cpp → c
    g++ → gcc
- Code organization/ Concepts / Implementation remains same
- No complex OOPs, no Templates, no C++ only thing…

Implementing Complex TCP Servers

We will go beyond simple implementation of client/servers programs present all over internet (Advanced Course)

Course objective : *Learn how to implement a typical complex Networking Socket library, closely tied to thread management*

Agenda

1. How to manage Multiple Clients through Multiplexing
2. Creating Multi-Threaded Clients
3. Forcefully disconnecting the client
4. Gracefully Shutting down TCP Server
5. Notifying events to application
    1. Client new connection
    2. Client disconnection
    3. Client msg recvd
6. Detecting connection live-ness using Keep Alive msgs
7. Handling Concurrency using locks
8. TCP Msg Demarcation
9. Maintaining statistics per client connection
10. Client Migration
11. Building Socket Programming C++ Library over Posix

Take Away :

1. Dividing a software design into multiple threads
2. Inter thread communication
3. Implementing Blocking Calls
4. Thread Synchronization using Semaphores, Mutexes
5. Scalable multi-threaded design
6. Understand how to write code which manage threads
7. System Design and Implementation Exercise
9. Real world meaningful project based on Thread Management
10. Decorate Resume with this fascinating project

Level Of Difficulty :

Intermediate level
Touches multiple concepts
(Managing Sockets, Multi-threading, Thread Sync, Byte Arrays etc)

Socket Programming Implementing Complex TCP Servers

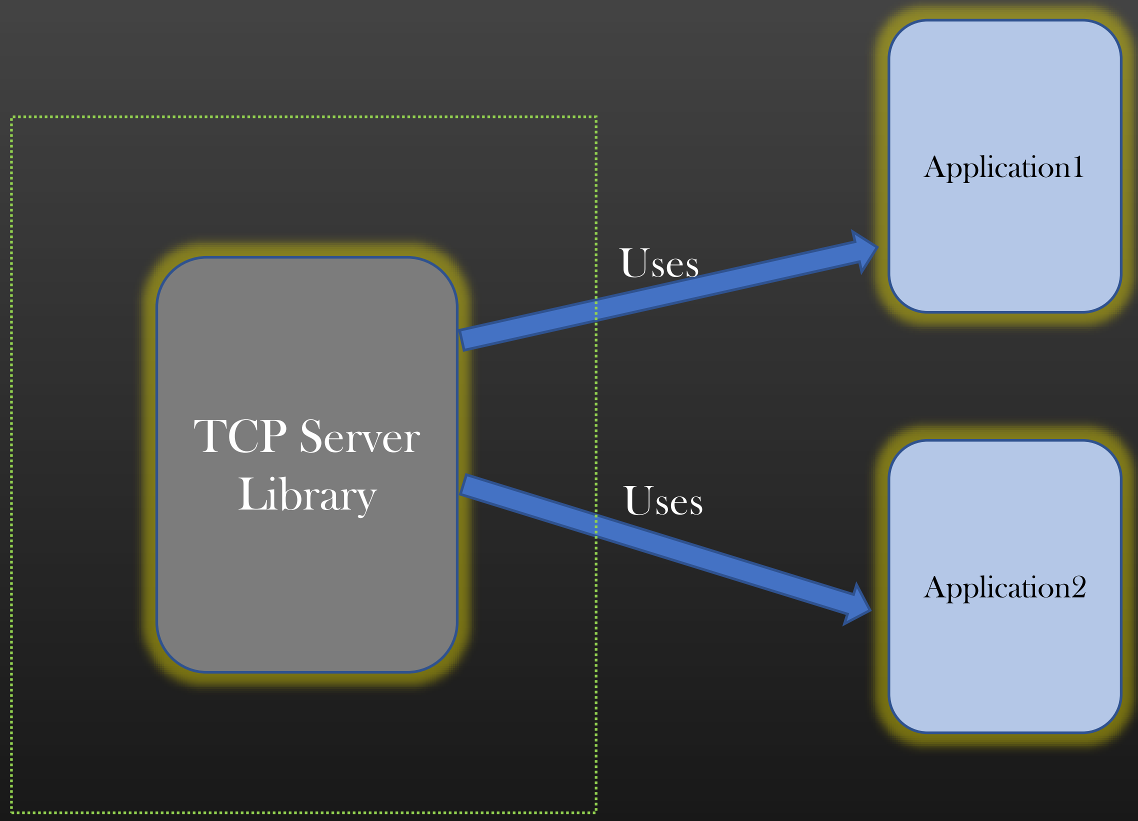
By CSEPracticals

Networking Operating Systems Linux System Programming Kernel Network Protocols TCP/IP
Memory Management IPC RPC Multi-threading Socket Programming Asynchronous Programming

www.csepracticals.com
Telegram grp : telecsepracticals

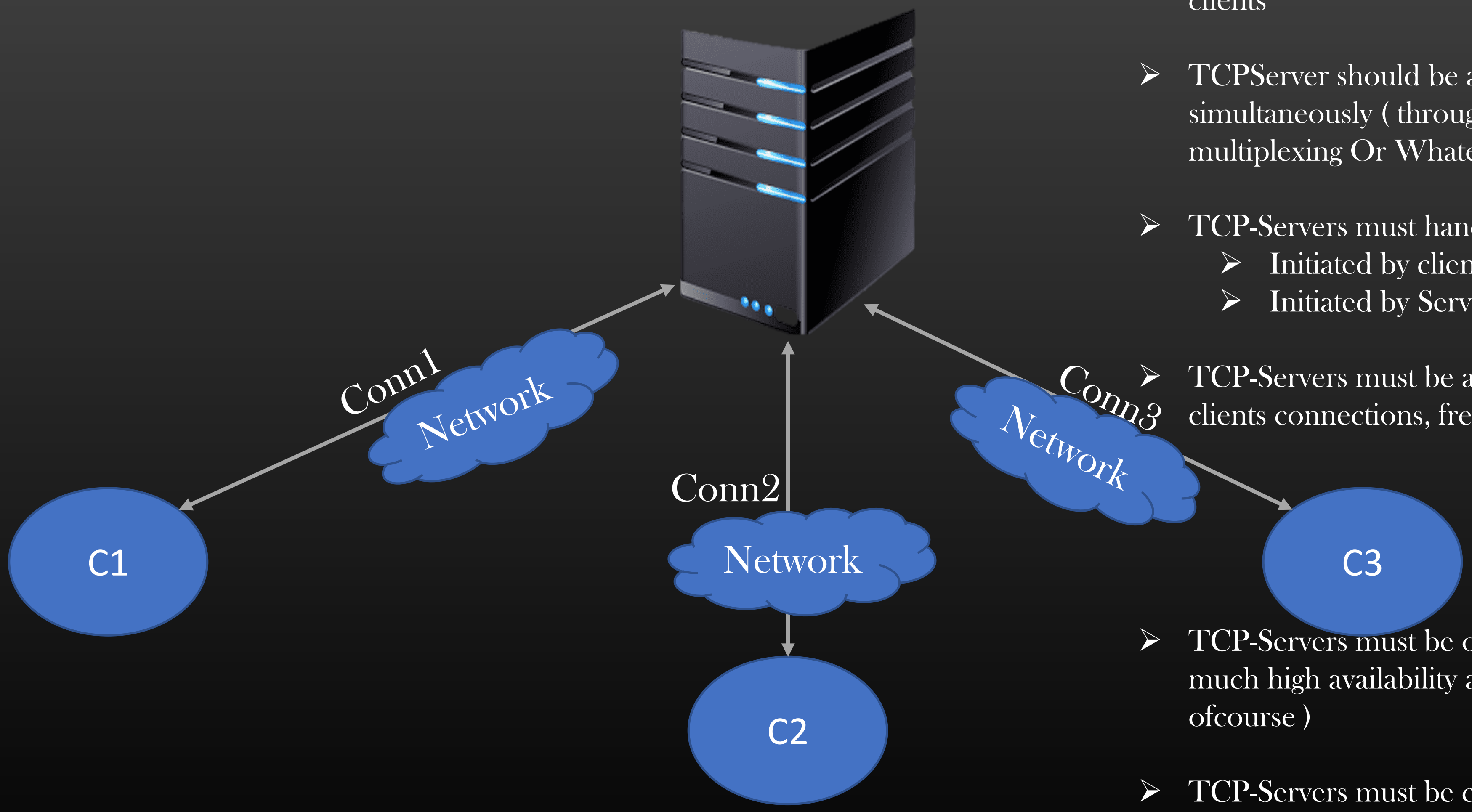
Codes : https://github.com/sachinites/TCPServerLib

Implementing Complex TCP Servers → End Product of this Course



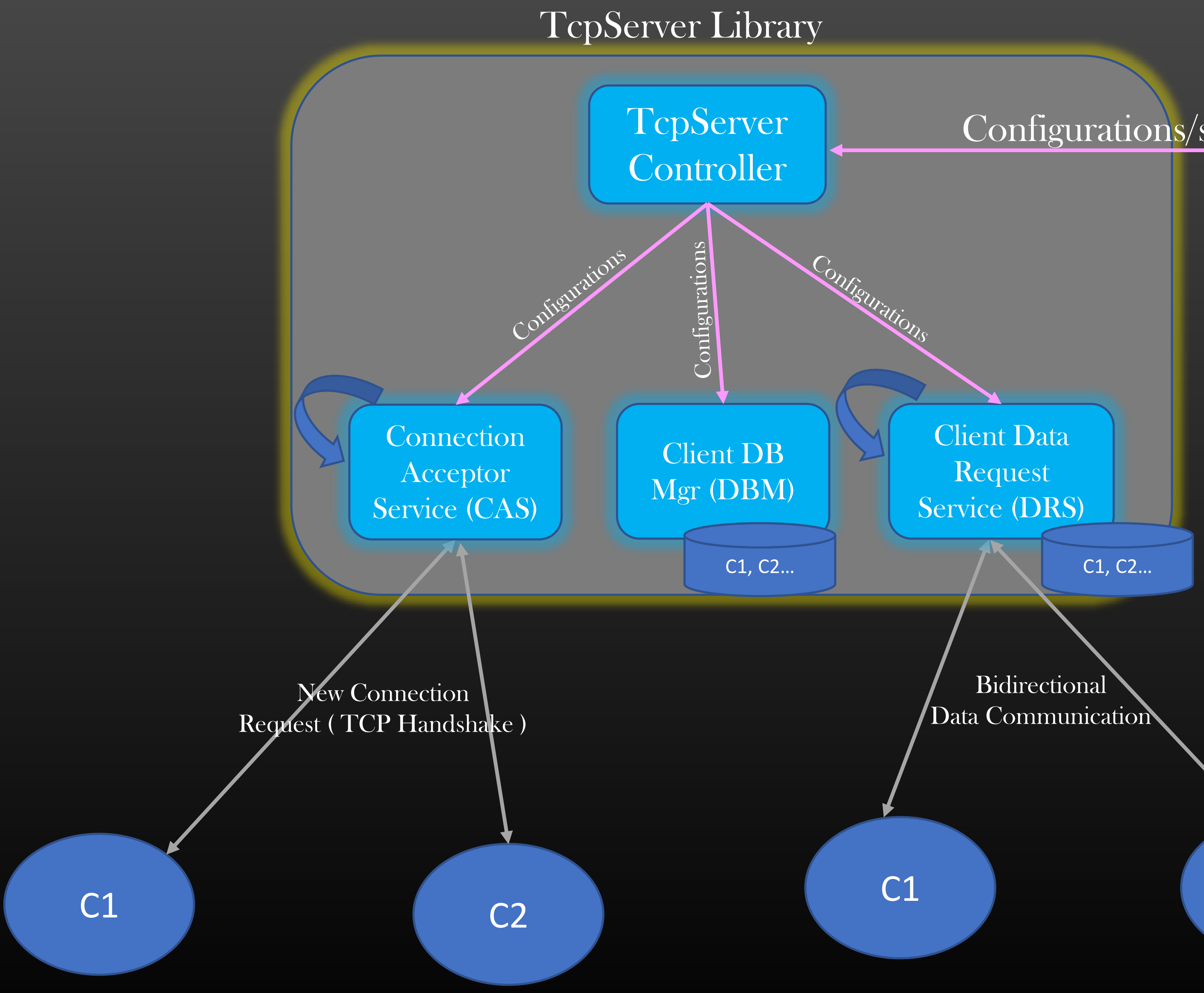
- Applns can create unlimited no of TCPServers
- Applns are notificed about client's
  - Connection
  - Discinnection
  - Msg Recvd
- Appln can incrementally build more protocols over
  TCP Server Library
  > Eg FTP Server

- TCP-Server Working :



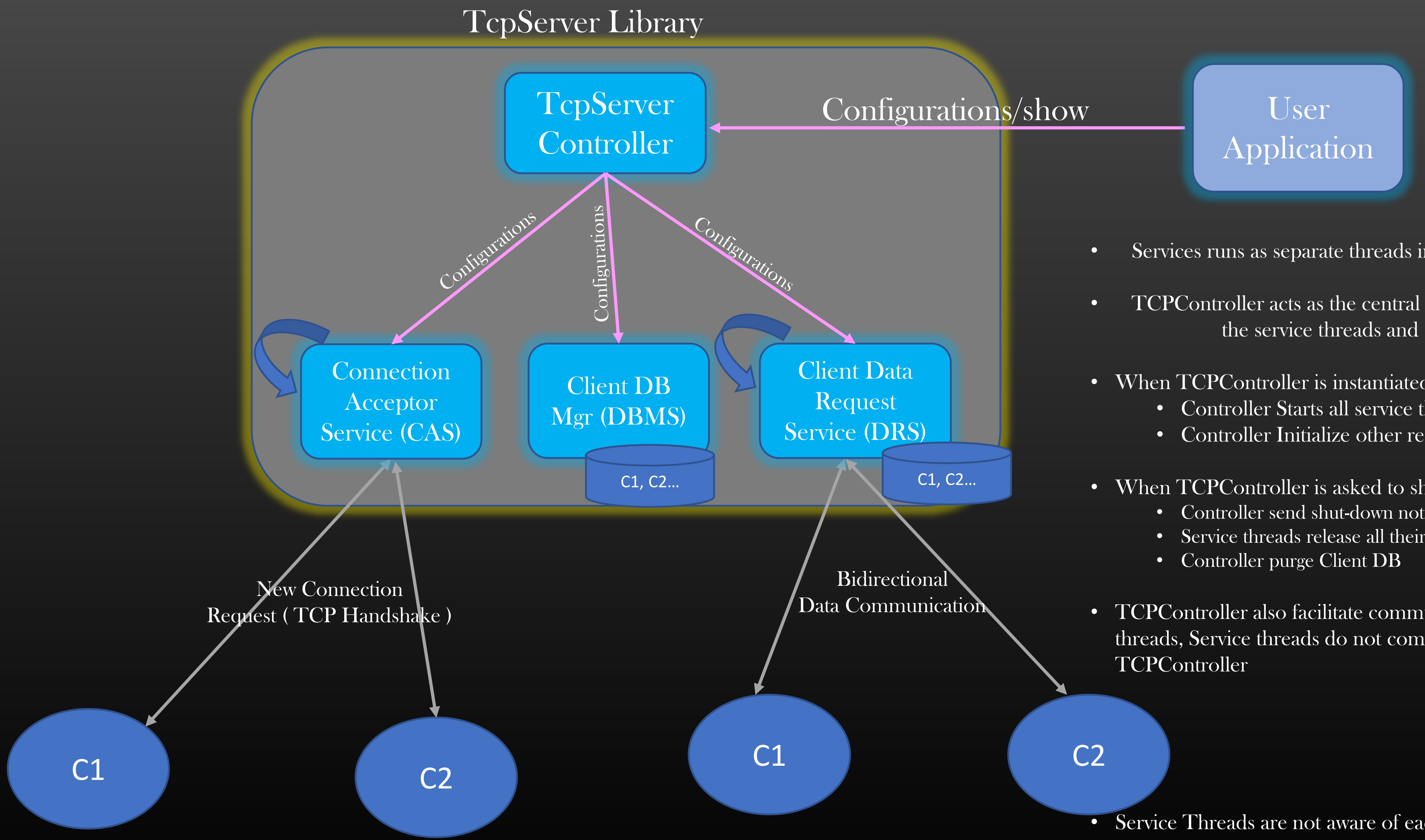
- TCP-Servers, On Starting, must listen on some user configured IP-Address and Port No
- TCPServers must be available to accept new connection requests from new clients
- TCPServer should be able to handle multiple Client's Connections simultaneously (through multi-threading Or multi-processing or multiplexing Or Whatever)
- TCP-Servers must handle connection disconnection
  - Initiated by client
  - Initiated by Server itself
- TCP-Servers must be able to shut-down gracefully (disconnecting all clients connections, free up all resources before terminating the process)
- TCP-Servers must be optimized to service Maximum Clients and with as much high availability as Possible (depending on machine capacity ofcourse)
- TCP-Servers must be configurable to abide by certain rules :
  - Accept no more new connections
  - Stop listening to all or particular client etc

Implementing Complex TCP Servers → Our Project Design Overview



- Controller
  - It is a Set of APIs
  - APIs invoked by Appln Thread / Service Threads
  - APIs to manage Service Operations
- CAS
  - Responsible to accept new connection Request (accept ())
  - It is a thread
- DBMS
  - Responsible to store all connected clients
  - It is a Data structure
- DRS
  - Responsible for Communication with Connected clients
  - It also maintain one more copy of all connected clients
  - It is a thread + Data Structure

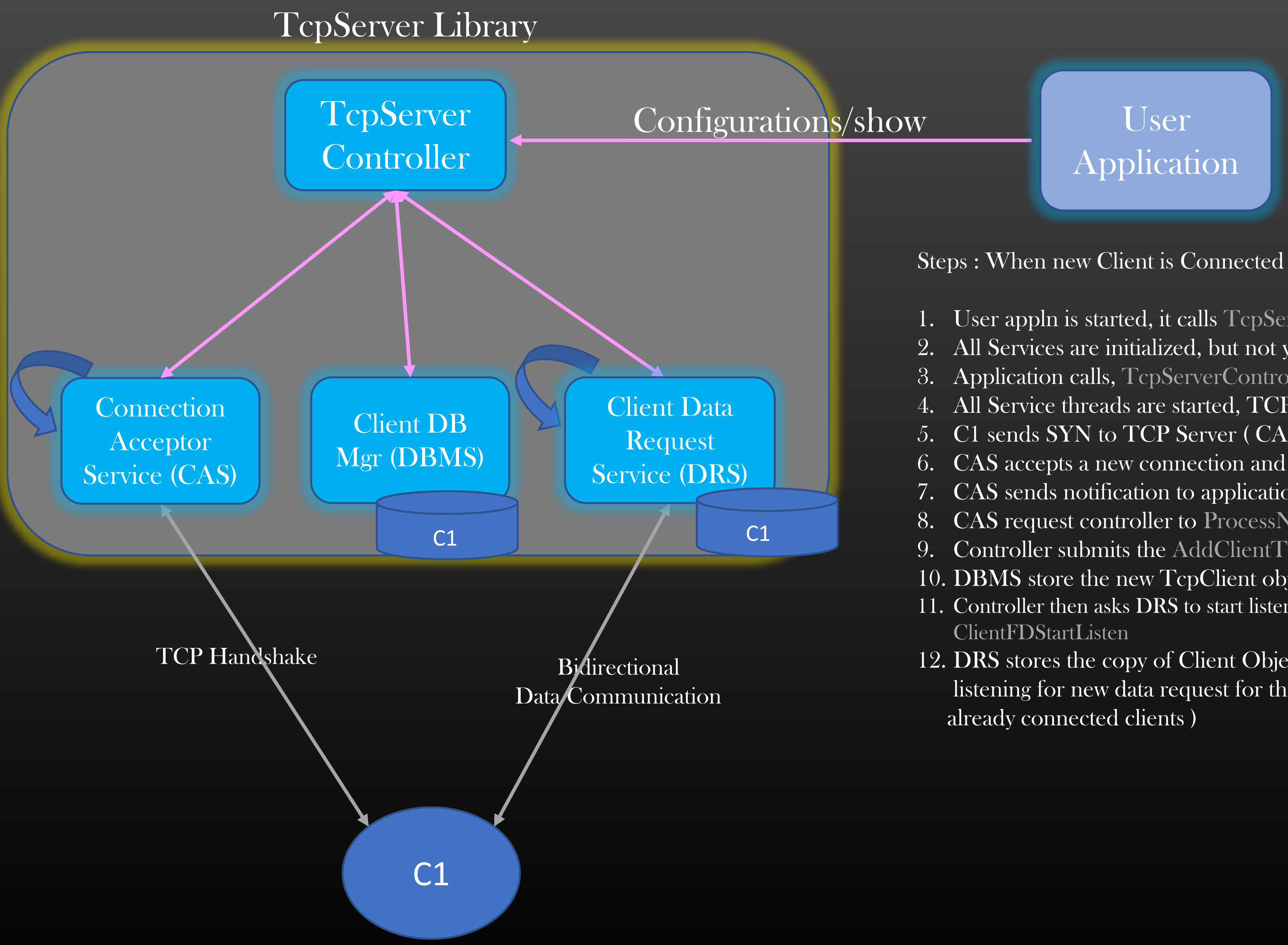
Implementing Complex TCP Servers → Our Project Design Overview



- Services runs as separate threads in infinite loop
- TCPController acts as the central entity responsible to manage the service threads and Client DB
- When TCPController is instantiated/Started by the application
  - Controller Starts all service threads
  - Controller Initialize other resources required
- When TCPController is asked to shut-down by application :
  - Controller send shut-down notification to all service threads
  - Service threads release all their resources before shutting down
  - Controller purge Client DB
- TCPController also facilitate communication between service threads, Service threads do not communicate directly but through TCPController
- Service Threads are not aware of each other. Simple Design, Scalable, Demarcation of Responsibilities, No class dependency

Implementing Complex TCP Servers → Getting Started with Implementation

Implementing Complex TCP Servers → New Client Connection (SYN) Processing Steps



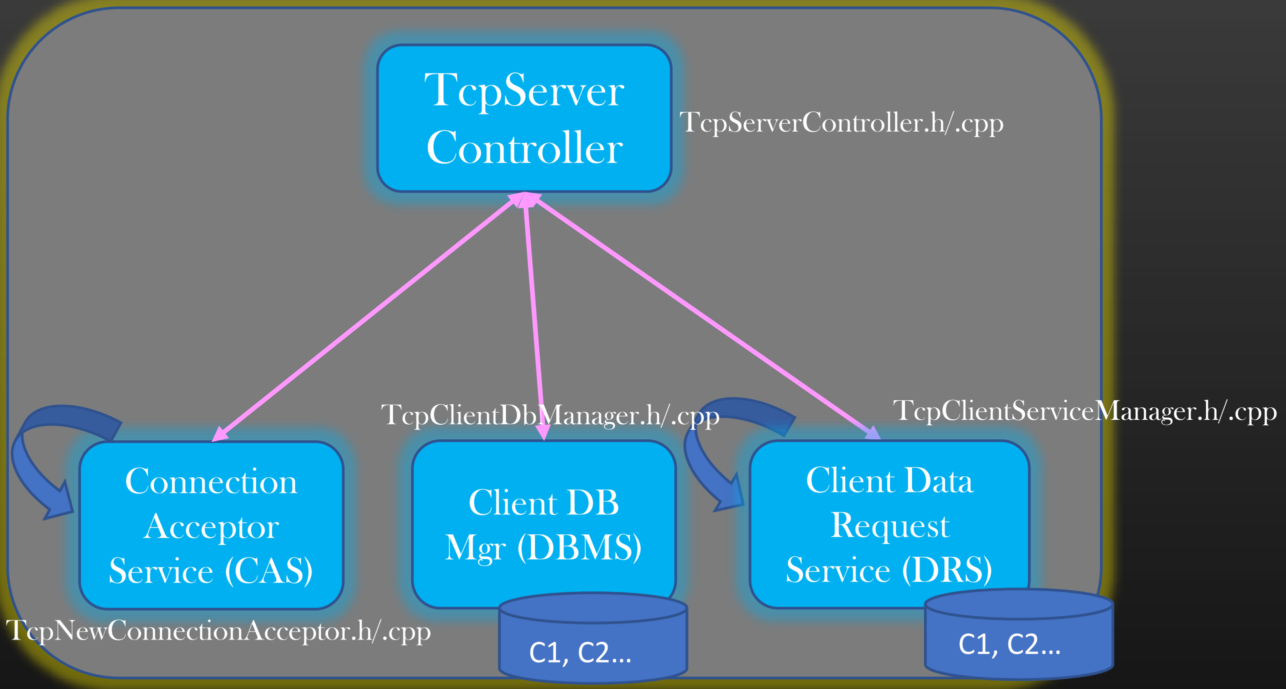
Steps : When new Client is Connected

1. User appln is started, it calls TcpServerController ()
2. All Services are initialized, but not yet running
3. Application calls, TcpServerController()->Start()
4. All Service threads are started, TCPServer is ready
5. C1 sends SYN to TCP Server (CAS)
6. CAS accepts a new connection and create TcpClient Object
7. CAS sends notification to application directly for new connection
8. CAS request controller to ProcessNewClient Object
9. Controller submits the AddClientToDB Request to DBMS
10. DBMS store the new TcpClient object in DB
11. Controller then asks DRS to start listen for this new Client Object ClientFDStartListen
12. DRS stores the copy of Client Object in its own local DB and start listening for new data request for this new Client (along with other already connected clients)

*Implementing Complex TCP Servers → Project Files Set Up

- Let us start with writing application file testapp.c and TcpServer Controller TcpServerController.h/.cpp

TcpServer Library



TcpServer Controller
TcpServerController.h/.cpp
TcpClientDbManager.h/.cpp
TcpClientServiceManager.h/.cpp
Connection Acceptor Service (CAS)
Client DB Mgr (DBMS)
Client Data Request Service (DRS)
TcpNewConnectionAcceptor.h/.cpp
C1, C2…
C1, C2…

git clone https://github.com/sachinites/TCPServerLib/

Dir : TCPServerLib/Course

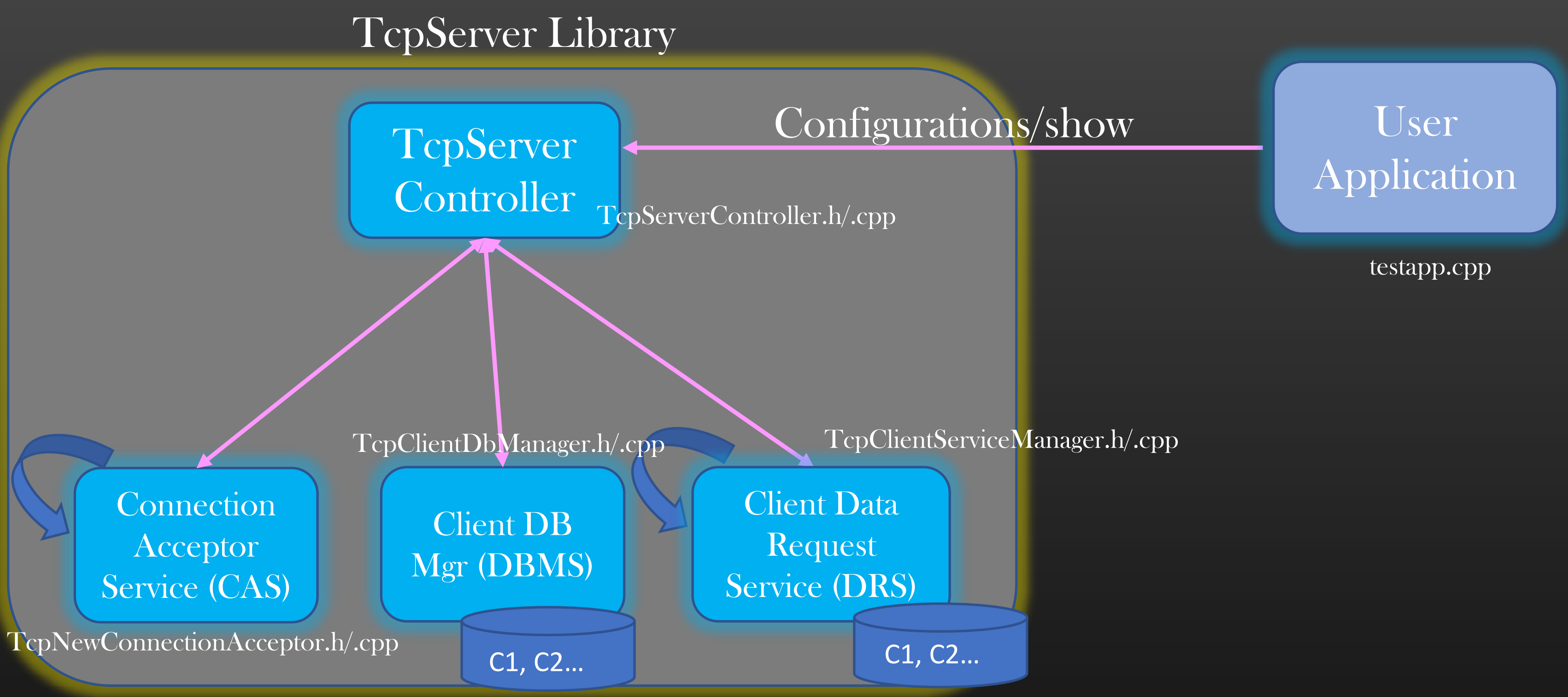
Dir : TCPServerLib

```
class TcpServerController {

    private:
        TcpNewConnectionAcceptor *tcp_new_conn_acc;
        TcpClientDbManager *tcp_client_db_mgr;
        TcpClientServiceManager *tcp_client_svc_mgr;

    public:
        uint32_t ip_addr;
        uint16_t port_no;
        std::string name;
        TcpServerController(std::string ip_addr,
                uint16_t port_no, std::string name);
        ~TcpServerController();
        void Start();
        void Stop();
} ;
```

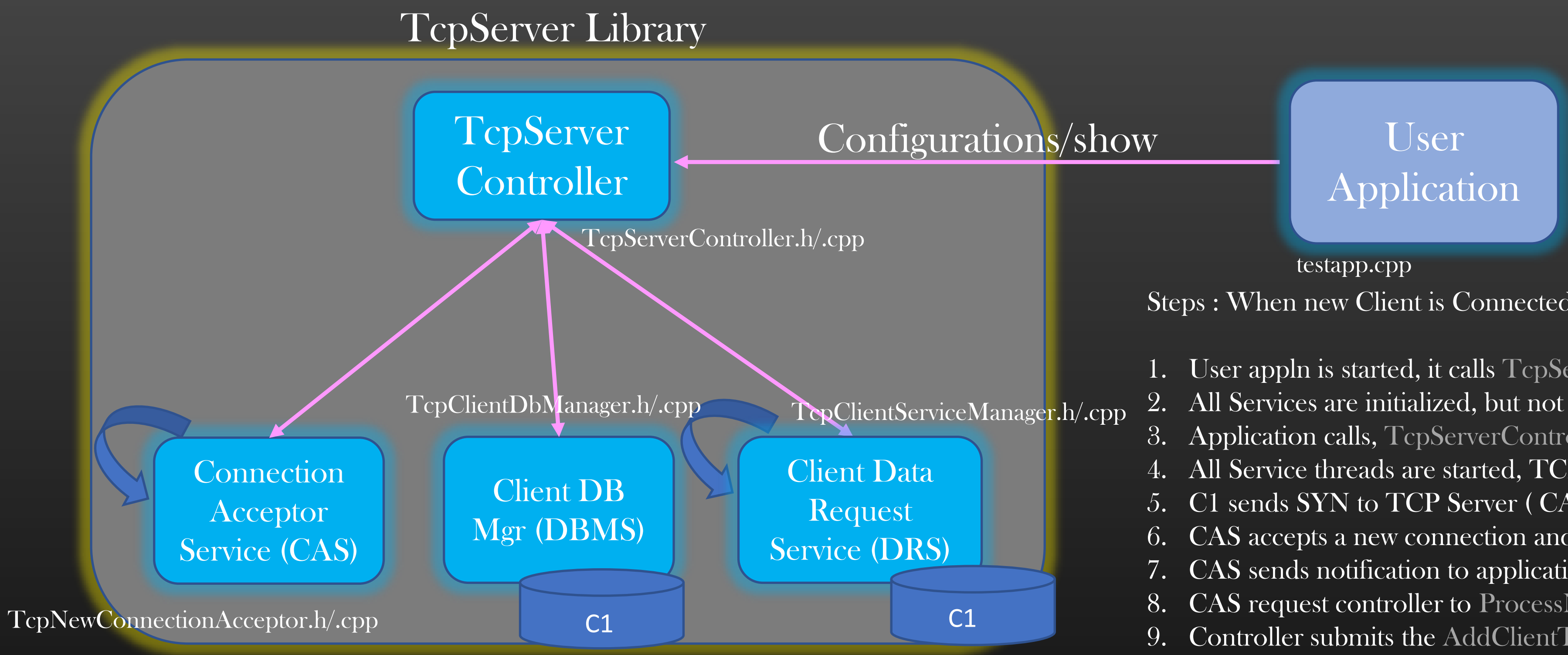
Implementing Complex TCP Servers → Getting Started with Implementation → Makefile Setup


TcpServer Library
TcpServer Controller
TcpServerController.h/.cpp
Configurations/show
User Application
testapp.cpp
TcpClientDbManager.h/.cpp
TcpClientServiceManager.h/.cpp
Connection Acceptor Service (CAS)
Client DB Mgr (DBMS)
Client Data Request Service (DRS)
TcpNewConnectionAcceptor.h/.cpp
C1, C2…
C1, C2…

☞ Let us Compile and build the executable

☞ Makefile Attached in the Resource Section

Implementing Complex TCP Servers → New Client Connection (SYN) Processing Steps


TcpServer Library
TcpServer Controller
Configurations/show
User Application
TcpServerController.h/.cpp
testapp.cpp
TcpClientDbManager.h/.cpp
TcpClientServiceManager.h/.cpp
Connection Acceptor Service (CAS)
Client DB Mgr (DBMS)
Client Data Request Service (DRS)
C1
C1
TcpNewConnectionAcceptor.h/.cpp

Steps : When new Client is Connected

1. User appln is started, it calls TcpServerController ()
2. All Services are initialized, but not yet running
3. Application calls, TcpServerController()->Start()
4. All Service threads are started, TCPServer is ready
5. C1 sends SYN to TCP Server (CAS)
6. CAS accepts a new connection and create TcpClient Object
7. CAS sends notification to application directly for new connection
8. CAS request controller to ProcessNewClient Object
9. Controller submits the AddClientToDB Request to DBMS
10. DBMS store the new TcpClient object in DB
11. Controller then asks DRS to start listen for this new Client Object ClientFDStartListen
12. DRS stores the copy of Client Object in its own local DB and start listening for new data request for this new Client (along with other already connected clients)

☞ Let us Start with the Implementation of our Project

Implementing Complex TCP Servers → TCP Client Data Structure

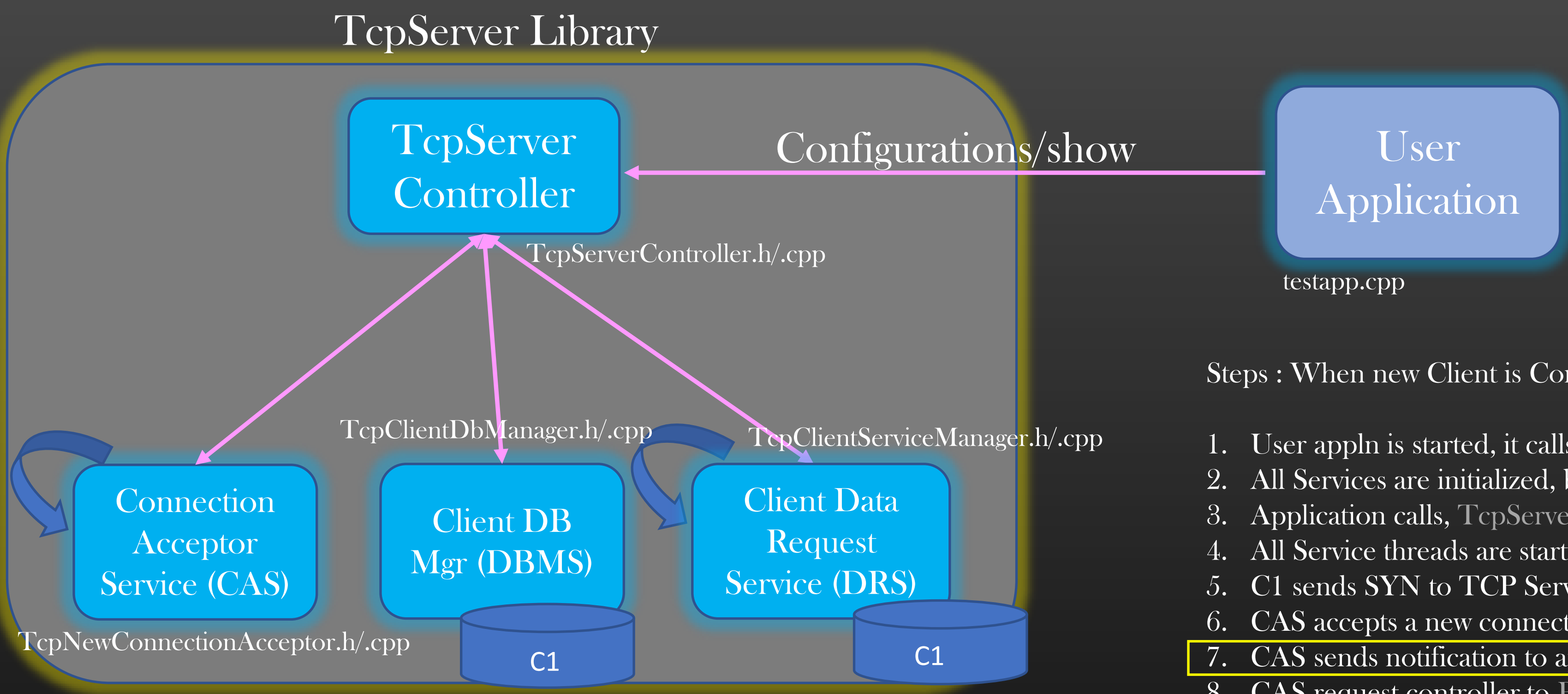
```
class TcpClient {

    private:
    public :
        uint32_t ip_addr;
        uint16_t port_no;
        int comm_fd;
        TcpServerController *tcp_ctrlr;
                TcpClient(uint32_t ip_addr, uint16_t port_no);
};
```

TcpClient.h/.cpp

☞ TCPServer maintains all connected clients using TcpClient Data Structure

Implementing Complex TCP Servers → Send Notification to UA

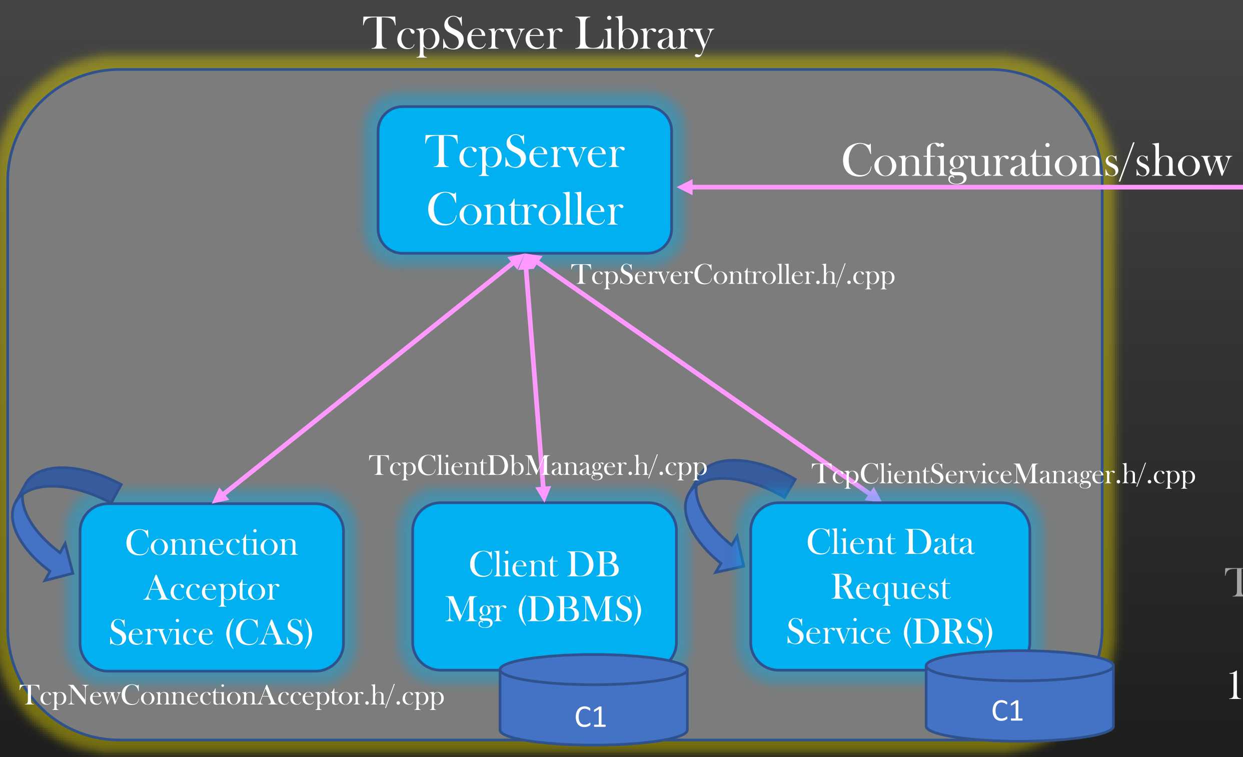

TcpServer Library
TcpServer Controller
TcpServerController.h/.cpp
Configurations/show
User Application
testapp.cpp
TcpClientDbManager.h/.cpp
TcpClientServiceManager.h/.cpp
Connection Acceptor Service (CAS)
Client DB Mgr (DBMS)
Client Data Request Service (DRS)
TcpNewConnectionAcceptor.h/.cpp
C1
C1

Steps : When new Client is Connected

1. User appln is started, it calls TcpServerController ()
2. All Services are initialized, but not yet running
3. Application calls, TcpServerController()->Start()
4. All Service threads are started, TCPServer is ready
5. C1 sends SYN to TCP Server (CAS)
6. CAS accepts a new connection and create TcpClient Object
7. CAS sends notification to application directly for new connection
8. CAS request controller to ProcessNewClient Object
9. Controller submits the AddClientToDB Request to DBMS
10. DBMS store the new TcpClient object in DB
11. Controller then asks DRS to start listen for this new Client Object ClientFDStartListen
12. DRS stores the copy of Client Object in its own local DB and start listening for new data request for this new Client (along with other already connected clients)

- UA registers callbacks with TCP Controller for
  - *Connection*
  - Disconnection (later)
  - Msg recvd (later)
- CAS invokes these callback for *Connection* when new client connects to TCP Server

Implementing Complex TCP Servers → Send Notification to UA


TcpServer Library
TcpServer Controller
TcpServerController.h/.cpp
Configurations/show
User Application
testapp.cpp
TcpClientDbManager.h/.cpp
TcpClientServiceManager.h/.cpp
Connection Acceptor Service (CAS)
Client DB Mgr (DBMS)
Client Data Request Service (DRS)
TcpNewConnectionAcceptor.h/.cpp
C1
C1

Three Steps :

1. Define Fn pointers as members of TcpServerController class

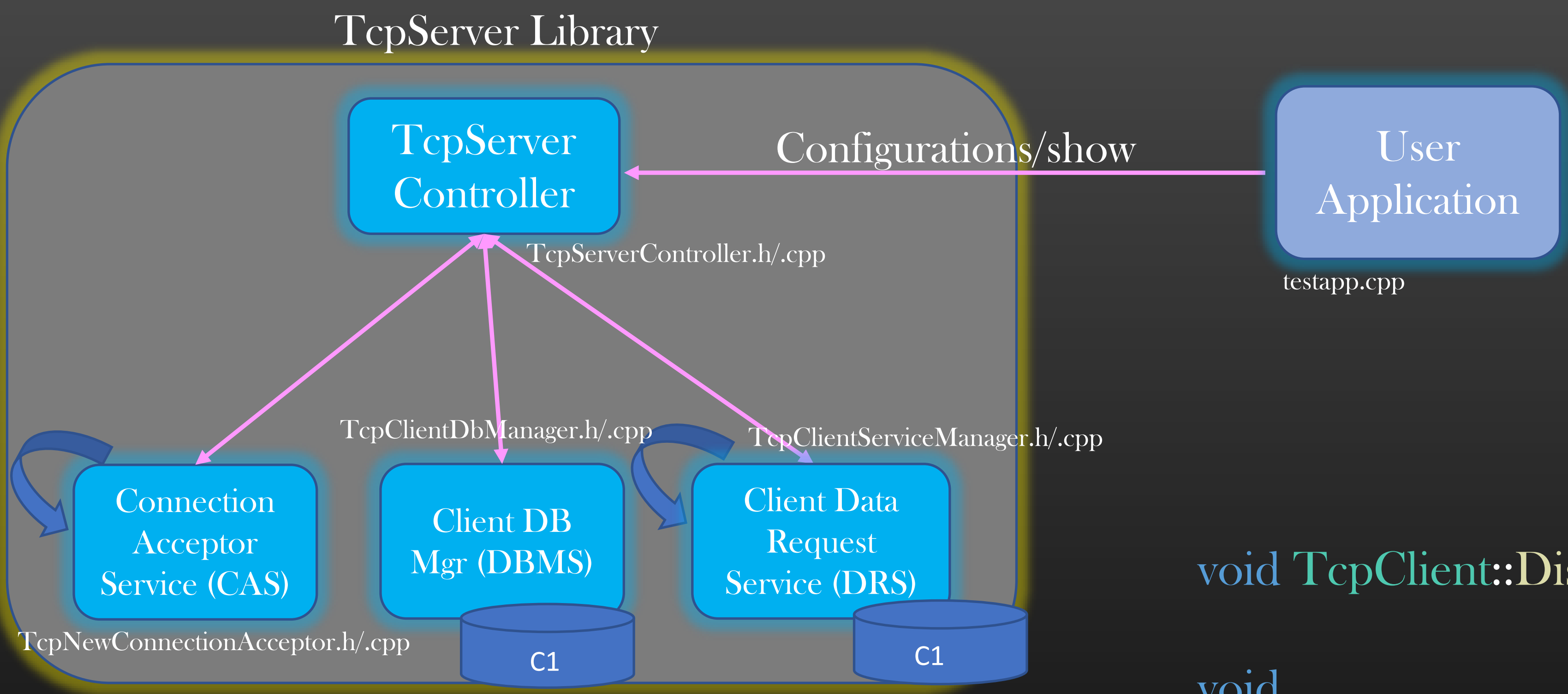
2. Application create callback fns and register with TcpServer Controller

3. CAS invokes the Appl's callback fn when Client connects to it

- UA registers callbacks with TCP Controller for
  - *Connection*
  - Disconnection (later)
  - Msg recvd (later)

- CAS invokes these callback for *Connection* when new client connects to TCP Server

Implementing Complex TCP Servers → Fn to display all Connected Client

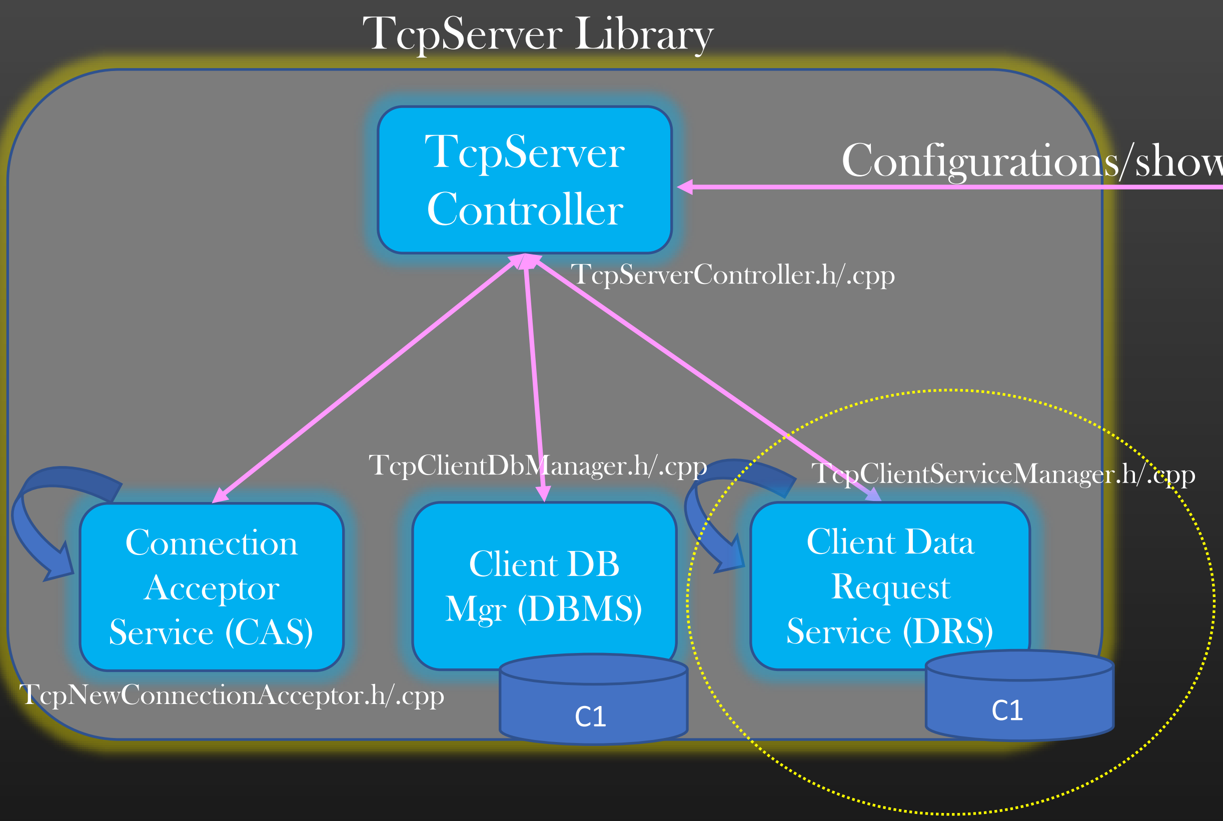

TcpServer Library
TcpServer Controller
TcpServerController.h/.cpp
Configurations/show
User Application
testapp.cpp
TcpClientDbManager.h/.cpp
TcpClientServiceManager.h/.cpp
Connection Acceptor Service (CAS)
Client DB Mgr (DBMS)
Client Data Request Service (DRS)
TcpNewConnectionAcceptor.h/.cpp
C1
C1

```
void TcpClient::Display() ;

void
TcpClientDbManager::DisplayClientDb();

void TcpServerController::Display() ;
```

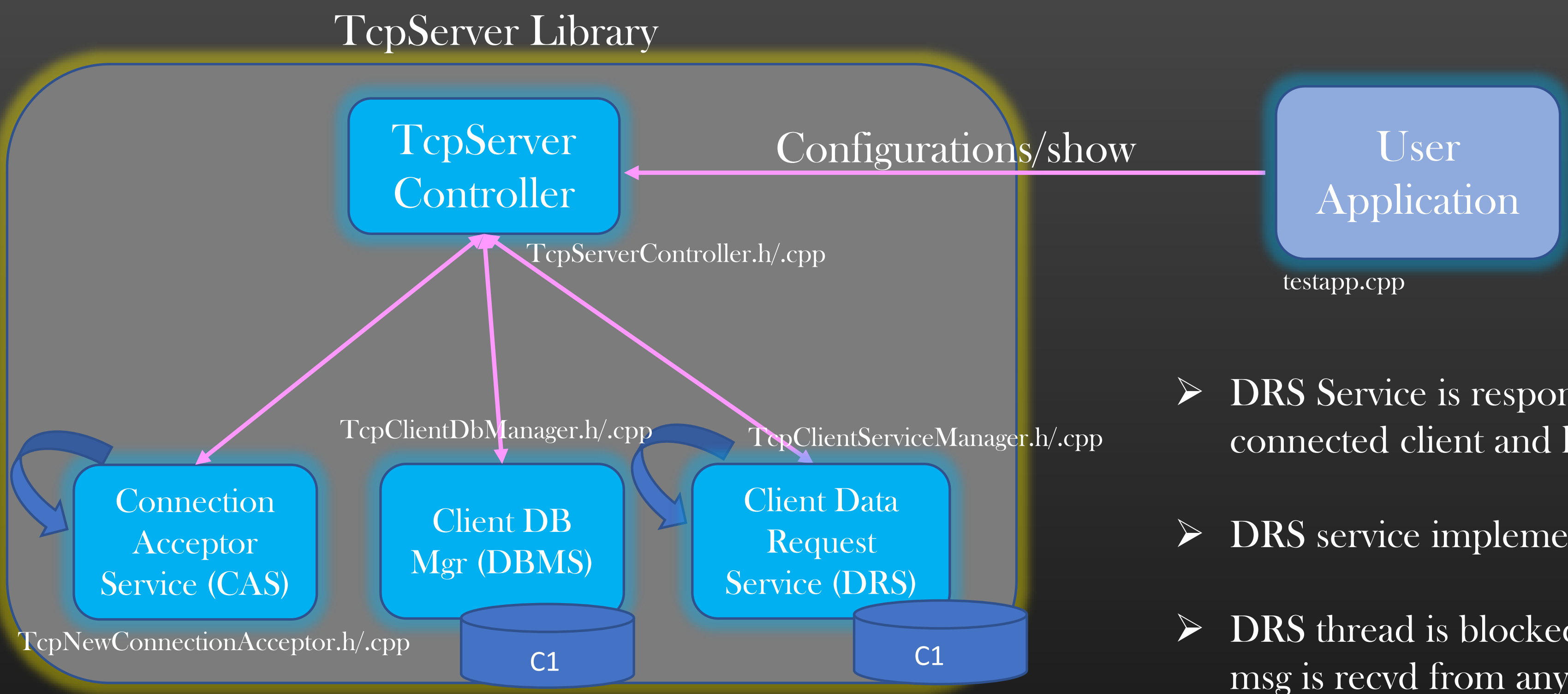
Implementing Complex TCP Servers


TcpServer Library
TcpServer Controller
TcpServerController.h/.cpp
Configurations/show
User Application
testapp.cpp
TcpClientDbManager.h/.cpp
TcpClientServiceManager.h/.cpp
Connection Acceptor Service (CAS)
Client DB Mgr (DBMS)
Client Data Request Service (DRS)
C1
C1
TcpNewConnectionAcceptor.h/.cpp

Implementing TCP Client Service Manager Thread

www.csepracticals.com
Telegram grp : telecsepracticals

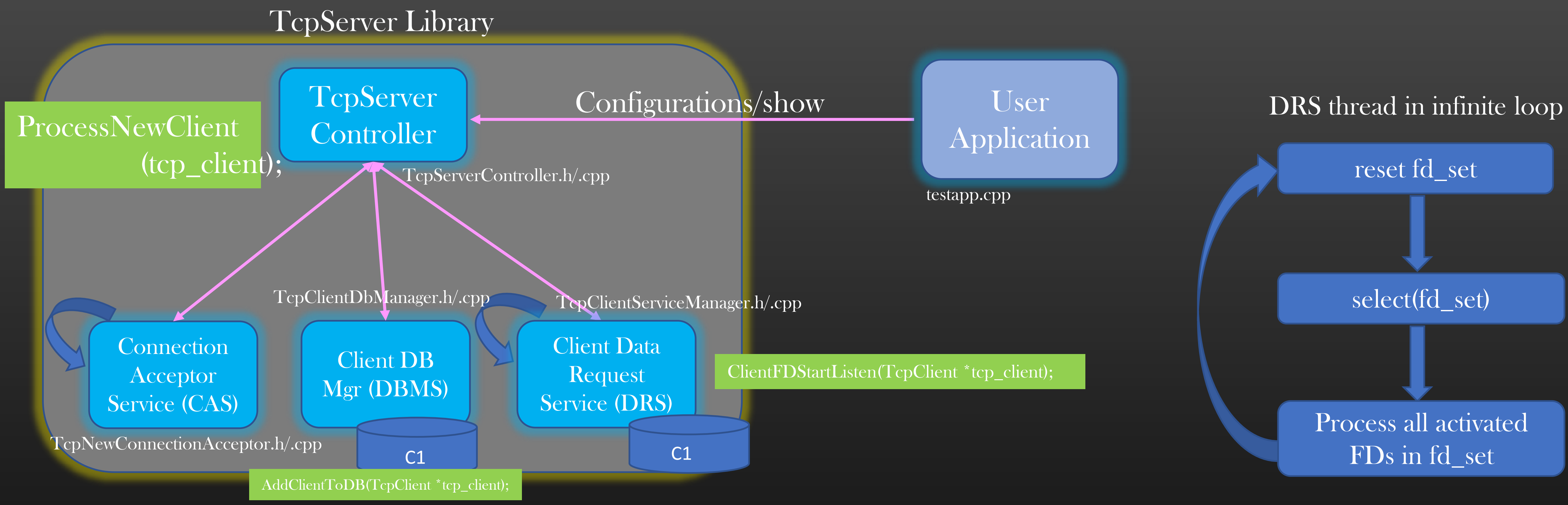
Implementing Complex TCP Servers → DRS Service


TcpServer Library
TcpServer Controller
Configurations/show
User Application
TcpServerController.h/.cpp
testapp.cpp
TcpClientDbManager.h/.cpp
TcpClientServiceManager.h/.cpp
Connection Acceptor Service (CAS)
Client DB Mgr (DBMS)
Client Data Request Service (DRS)
TcpNewConnectionAcceptor.h/.cpp
C1
C1

Pre-requisite : You know how select() works

- DRS Service is responsible for receiving messages from connected client and handover them to appln for processing
- DRS service implements select()/epoll() in a DRS thread
- DRS thread is blocked on select(), and unblocked as soon as msg is recvd from any client
- DRS is also called as *Tcp Client Service Manager*
- *Like CAS thread,* DRS thread is also started when TCPController Starts
- DRS maintains a separate copy of client database. A client Object is added to it by TCPController (next slide)

Implementing Complex TCP Servers → DRS Service → Adding a new Client FD to DRS


TcpServer Library
TcpServer Controller
ProcessNewClient (tcp_client);
Configurations/show
User Application
testapp.cpp
TcpServerController.h/.cpp
DRS thread in infinite loop
reset fd_set
select(fd_set)
Process all activated FDs in fd_set
TcpClientDbManager.h/.cpp
TcpClientServiceManager.h/.cpp
Connection Acceptor Service (CAS)
Client DB Mgr (DBMS)
Client Data Request Service (DRS)
ClientFDStartListen(TcpClient *tcp_client);
TcpNewConnectionAcceptor.h/.cpp
C1
C1
AddClientToDB(TcpClient *tcp_client);

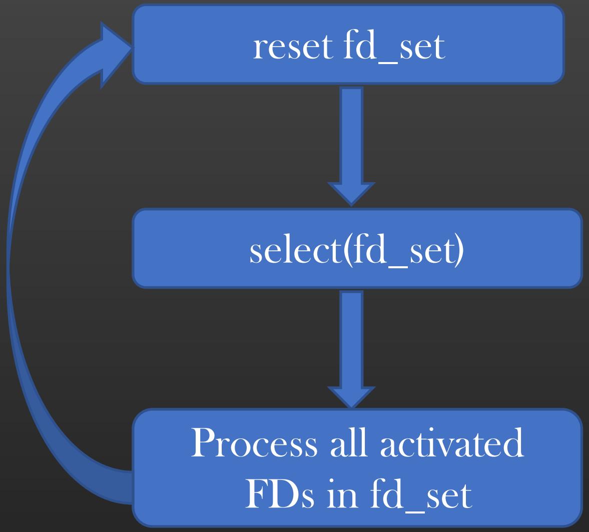
- ➢ TCP Controller hand-over the Client FD (generated by CAS thread) to DRS thread for listening
  TcpClientServiceManager::ClientFDStartListen(TcpClient *);

- ➢ Though ClientFDStartListen() will be invoked only when new client connects to CRS , Meanwhile, DRS thread could be in one of the following states :
  - ➢ blocked on select() Or
  - ➢ processing Client’s messages

Implementing Complex TCP Servers → DRS Service → Starting a DRS thread

```
class TcpClientServiceManager{

        . . .
        int max_fd;
        fd_set active_fd_set;
        fd_set backup_fd_set;
        pthread_t *client_svc_mgr_thread;
        std::list<TcpClient *>tcp_client_db;
        . . .
}
```



- When DRS thread is starting, we would not have any Connected clients (TCPServer is still staring its service threads ..)
- Include All Connected Client FDs present in DRS's Client DB in Multiplexing in a For loop

Implementing Complex TCP Servers → DRS Service → Stopping/Cancelling a DRS thread

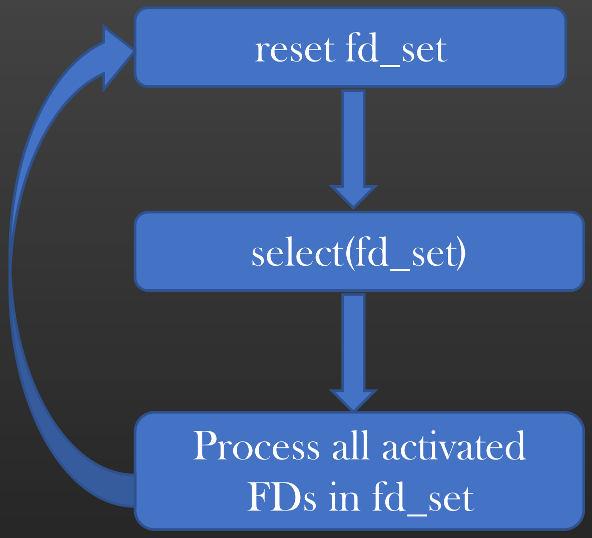
- We would need to stop/cancel the DRS thread
  - For example, Shutting down TCP Server, etc. ..

- What are the things in mind one should keep in mind to perform thread Cancellation ?

- One must cancel the running thread at *cancellation points* only
  - select()/epoll() is an inbuilt cancellation point

Public API to cancel the DRS thread :

```
void
TcpClientServiceManager::StopTcpClientServiceManagerThread();
```



DRS thread

Implementing Complex TCP Servers → DRS Service → Adding a new client FD to select()

Problem Statement :

CAS Thread wants DRS Thread to start listening on new Client FD

DRS thread could be in any state :

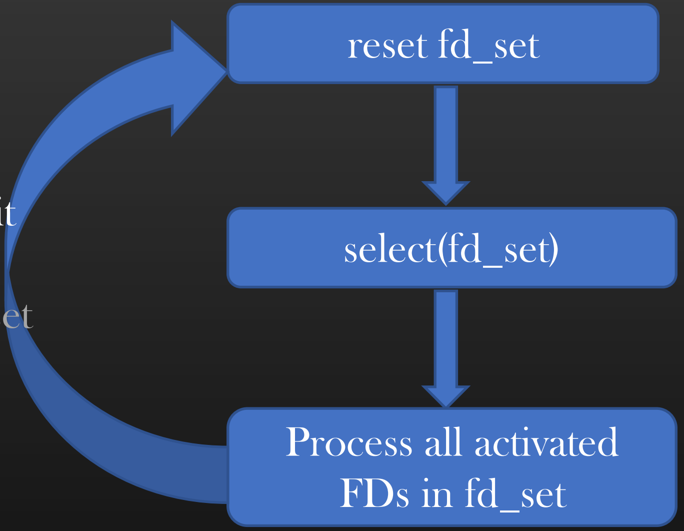
1. Blocked on select()
2. Servicing client's in a for loop

Challenge :

☞ if DRS thread is blocked on select, we cannot modify active_fd_set since it being used by select()

☞ if DRS thread is servicing clients in a for loop, we cannot modify the active_fd_set since it is being read by DRS thread (Read – Write Conflict)

DRS thread in infinite loop



Solution :

```
ClientFDStartListen(TcpClient *tcp_client) {
CAS Thread Cancels the DRS thread at Cancellation Points (pthread_cancel ())
CAS thread Waits for the Cancellation to complete (pthread_join() )
CAS thread Update DRS's Client DB
CAS thread Restart the DRS Thread
}
```

Implementing Complex TCP Servers → DRS Service → Adding a new client FD to select()

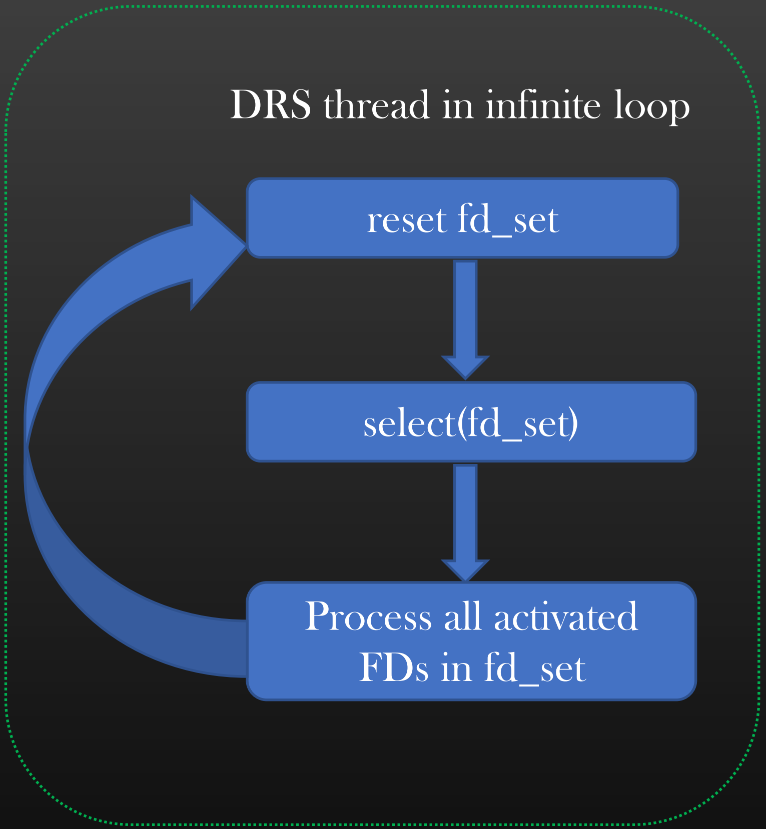
- Suppose DRS thread is blocked on select() monitoring clients – say 7 & 8

- Meanwhile CAS thread accept a new connection, and generate a new Client Comm FD – say 9

- CAS thread invokes via TCPController
  TcpClientServiceManager::
  ClientFDStartListen(TcpClient *);

Add a new Client FD to DRS client DB
Update *this->max_fd* if required
Add a new Client FD to active_fd_set

- Note that, ClientFDStartListen() is called in the context of CAS thread. Challenge here is CAS thread is trying to update the data structures which is being constantly read by DRS thread in infinite loop (Concurrency Issues !!)


DRS thread in infinite loop
reset fd_set
select(fd_set)
Process all activated FDs in fd_set

https://stackoverflow.com/questions/42501437/adding-new-fds-to-fd-set-while-blocking-on-select
https://stackoverflow.com/questions/9999801/add-remove-socket-descriptors-from-poll?rq=1
(Resource Section)

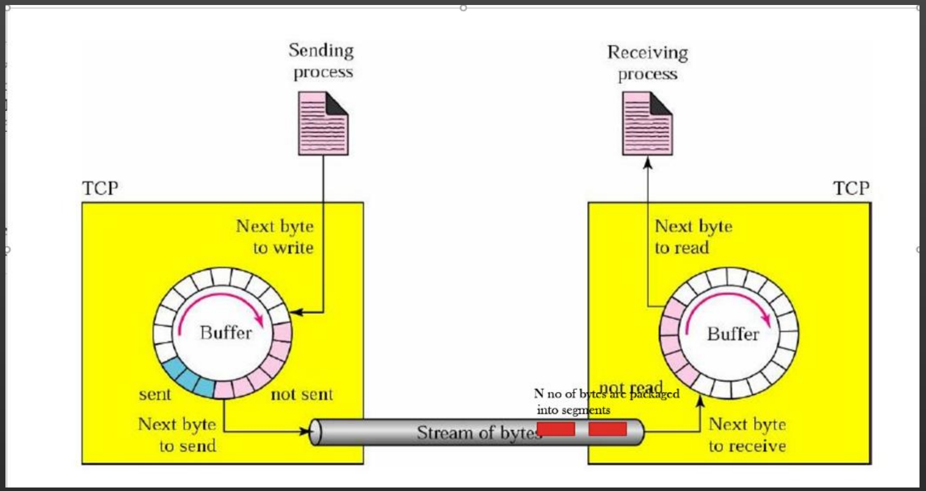
TCP-Message Demarcation

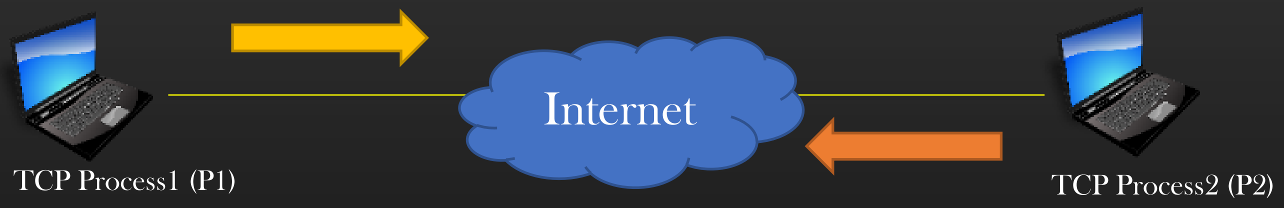
Implementing Complex TCP Servers → Byte Oriented Protocol

- TCP is a byte-oriented protocol

- It sees data as stream of bytes, it recognizes no start or end of msg in a stream of bytes

- Like flow of water





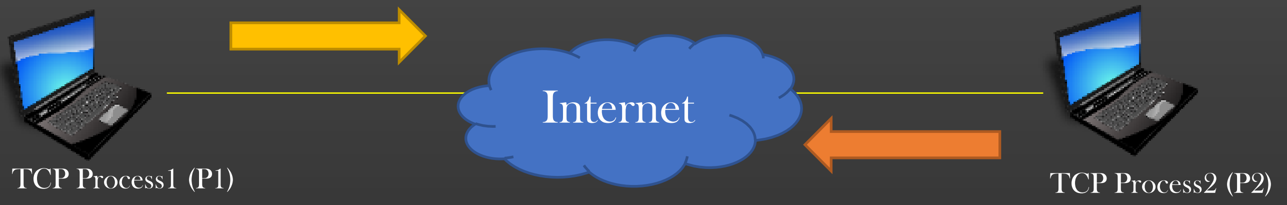
- Lets say application on TCP process1 send msg “Hello Abhishek” to TCP process2
- It is not necessary the P1 will send the entire msg to P2 in just one segment, Lets say MSS is set to 4B
- P1 will send the following segments –
  [Hell]
  [o Ab]
  [hish]
  [ek]
- P2 will recv 4 segmens in order

The Recipient application has no way to find if sending TCP sent 4 msgs or 1 msg ! ☠

Implementing Complex TCP Servers → Byte Oriented Protocol

```
        struct student {

        char name[128];
        int rollno;
        char address[256];
};
```



```
struct student stud;
. . .
sendto (&stud);
```

```
recv(buffer);
. . .
struct student *stud =
            (struct student *)buffer


Printf (stud->name);
Printf (stud->rollno);
Printf (stud->address)
```

This will fail if TCP delivers the Msg in smaller chunks to application

- Thus, TCP does not know where the msg begins and where it ends
- All it knows is that msg is sequence of bytes
- This problem is difficult to reproduce for smaller msgs, but immediately reproducible for larger msgs

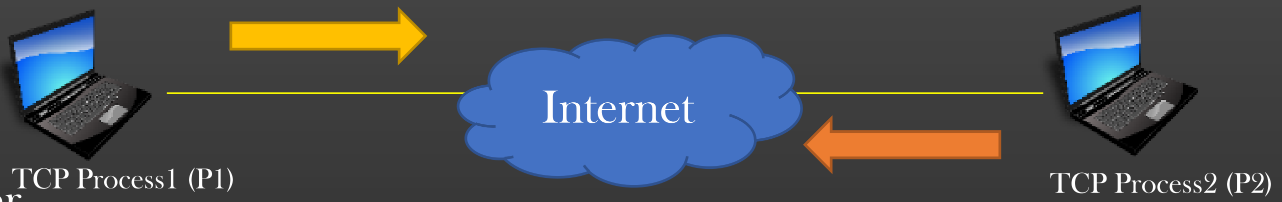
- So, Question is how TCP can be used to exchange fixed size messages, like most applications do

- Lets see one more scenario

Implementing Complex TCP Servers → Byte Oriented Protocol

- TCP can also do opposite

- If P1 sends multiple msgs in a loop, msgs can be unpredictably assembled together



- Demo : tcp_client_string_sender.cpp

https://www.codeproject.com/Articles/11922/Solution-for-TCP-IP-client-socket-message-boundary

Implementing Complex TCP Servers → TCP Message Demarcation

- Often we need that TCP peers exchange messages of known size, but given the TCP byte-oriented nature, we can not be sure if entire msg is delivered to recipient or in chunks Or assembled
  - Downloading 1GB file usually results in invoking recvfrom( ) many times (Splitting)
  - Sending smaller individual msgs at a high rate may results in concatenation of msgs

- Unless the recipient application is made intelligent to recognize the boundary of the msg, application cannot process the msg, splitted or assembled msgs are junks for an application which expects a fixed size message

  - TCP downloader and uploader works smoothly with TCP being byte oriented , no message boundaries recognition is required

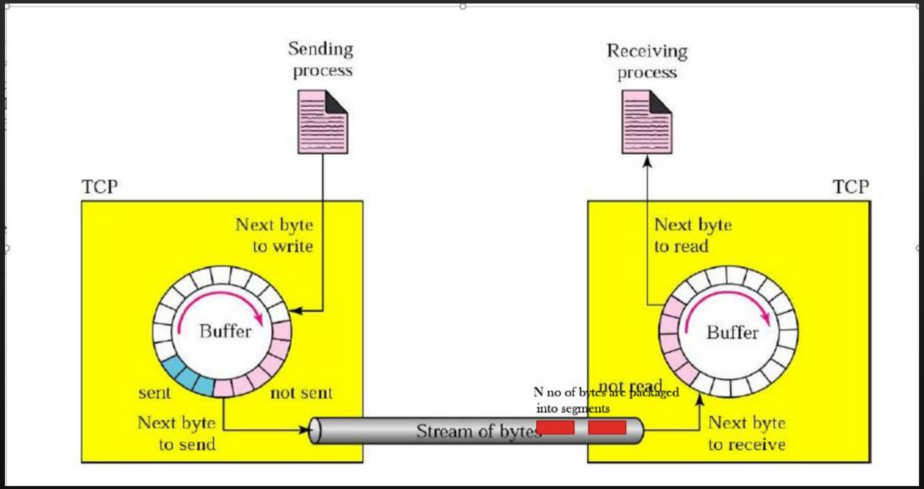
  - Email Client – Need to download several emails from Email Server, need to recognize message boundaries to identify each individual email

- We cannot modify the TCP protocol behavior, it is implemented this way

- We would need to make our application intelligent
  - Solution lies at application layer, not at TCP layer

Implementing Complex TCP Servers → TCP Message Demarcation →Fixed Msg Size Solution

- TCP Message Demarcation is a technique which makes the application aware of the message boundaries
- Until the application recvs a complete msg, application buffer the data
- As soon as application recvs required number of bytes of data, application remove the data from buffer and process it
- Soln : Maintain a Circular buffer at application layer



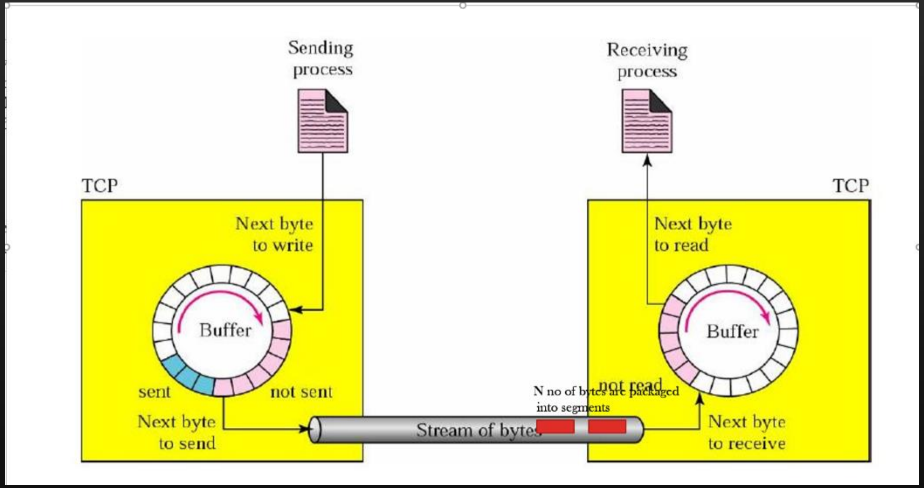
- This is fixed size msg solution, where recipient application is hard-coded with fixed size msg
- What if the recipient application need to process variable size data ?

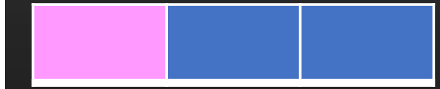
Implementing Complex TCP Servers → TCP Message Demarcation →Variable Msg Size Solution

- Variable Size data :
- Size of the msg is appended in the 2B hdr of the msg payload

10	4	2

Msg send by TCP sender





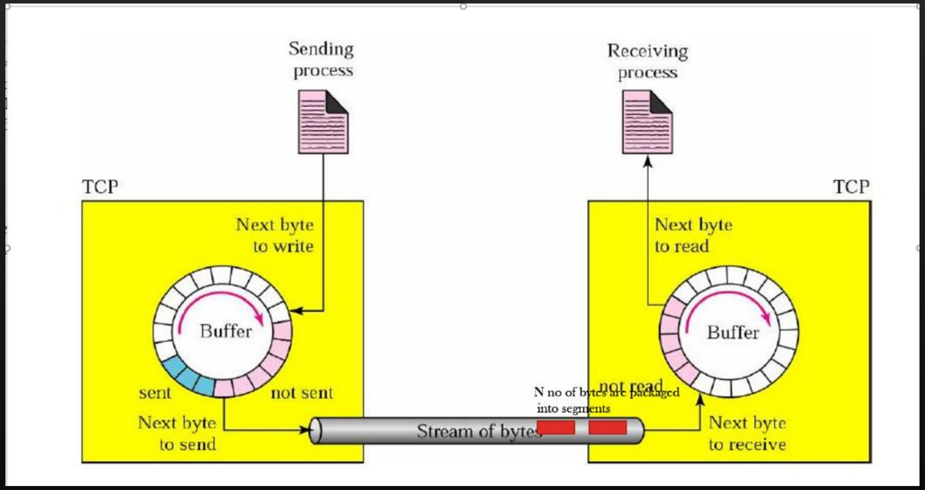
Application TCP circular buffer

Implementing Complex TCP Servers → TCP Message Demarcation

- Variable Size data :
- Size of the msg is appended in the 2B hdr of the msg payload

10	4	2	14	3	4	6	14	7	2	4	10	4	3

Snapshot of the msg accumulated in the recipient TCP Circular buffer



Recipient Application TCP circular buffer

Circular Buffer Implementation

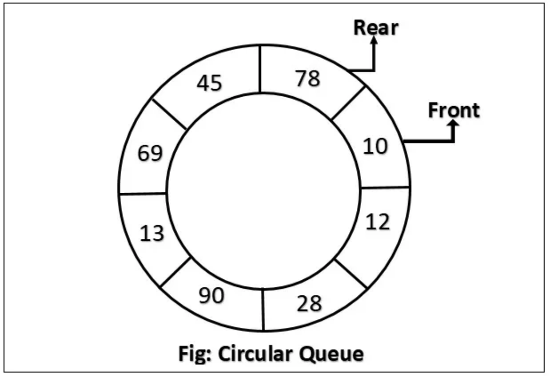
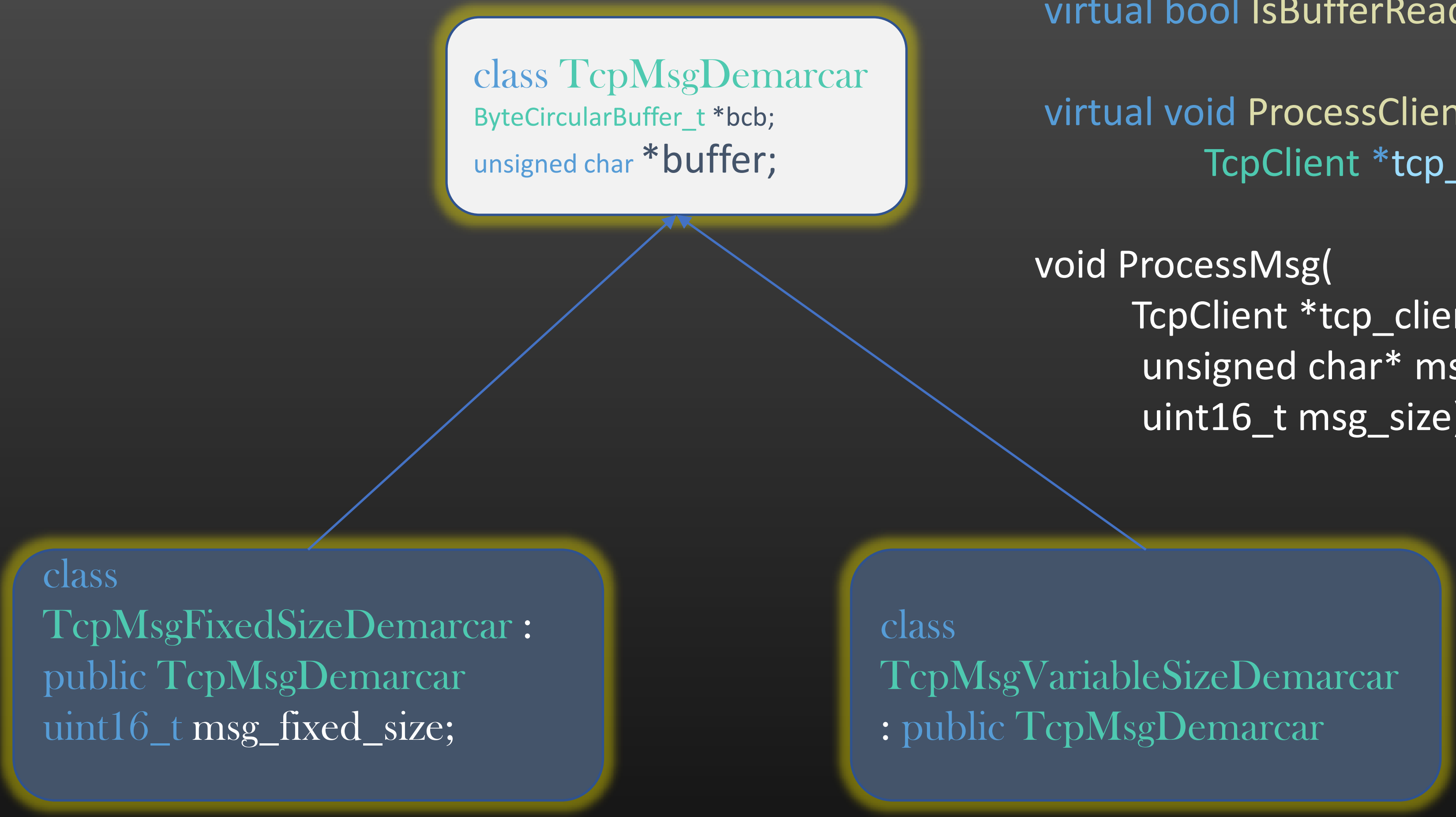


Fig: Circular Queue

- A Circular buffer is a Data structure which is nothing but a circular queue of bytes
- It has front and rear pointer
  - New bytes are queues at rear
  - Old bytes are removed from front
- We will implement BCB using pure C (don't use c++ specific things), though file name is .cpp
- Implementation :
  https://github.com/sachinites/TCPServerLib
  Files : ByteCircularBuffer.h/.cpp
- Let me walk you through the hdr file :
  - Either you do your own implementation
  - Or understand header file interface, and use the existing one in project directly
- Warning :
  - If you are going for your own implementation, integrate it with the TCP project after thorough testing
  - Else debugging will be a nightmare, we are working at byte level !

Implementing Complex TCP Servers → TCP Message Demarcation



class TcpMsgDemarcar
ByteCircularBuffer_t *bcb;
unsigned char *buffer;

class
TcpMsgFixedSizeDemarcar :
public TcpMsgDemarcar
uint16_t msg_fixed_size;

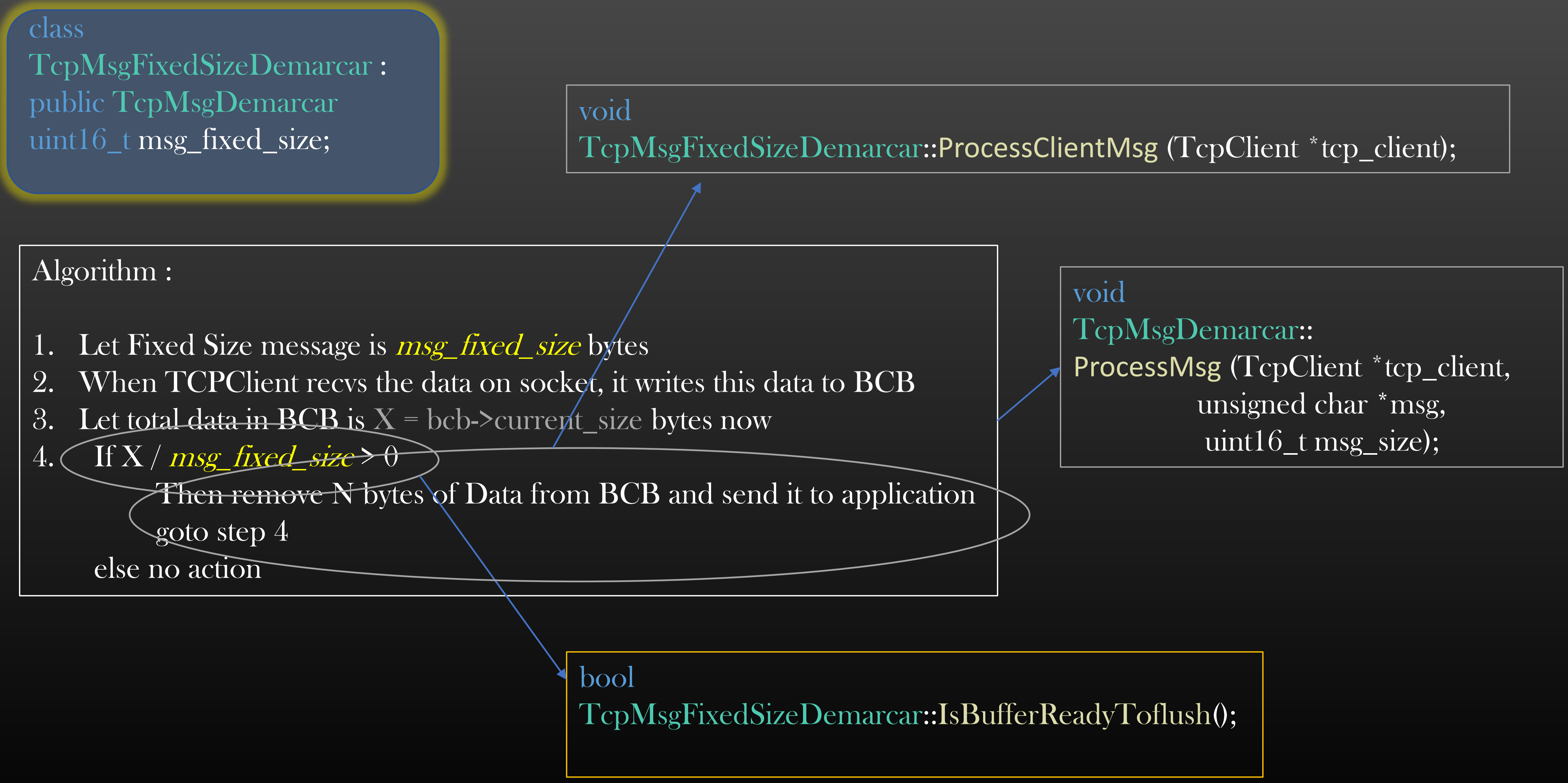
class
TcpMsgVariableSizeDemarcar
: public TcpMsgDemarcar

```
virtual bool IsBufferReadyToflush() = 0;

virtual void ProcessClientMsg(
            TcpClient *tcp_client) = 0;

void ProcessMsg(
        TcpClient *tcp_client,
         unsigned char* msg_recvd,
         uint16_t msg_size);
```

Implementing Complex TCP Servers → TCP Message Demarcation → TcpMsgFixedSizeDemarcar



```
class
TcpMsgFixedSizeDemarcar :
public TcpMsgDemarcar
uint16_t msg_fixed_size;
```

```
void
TcpMsgFixedSizeDemarcar::ProcessClientMsg (TcpClient *tcp_client);
```

```
Algorithm :

1.  Let Fixed Size message is msg_fixed_size bytes
2.  When TCPClient recvs the data on socket, it writes this data to BCB
3.  Let total data in BCB is X = bcb->current_size bytes now
4.    If X / msg_fixed_size > 0
          Then remove N bytes of Data from BCB and send it to application
          goto step 4
      else no action
```

```
void
TcpMsgDemarcar::
ProcessMsg (TcpClient *tcp_client,
              unsigned char *msg,
               uint16_t msg_size);
```

```
bool
TcpMsgFixedSizeDemarcar::IsBufferReadyToflush();
```

Integrating CLI Interface

Implementing Complex TCP Servers → Integrating CLI Interface

- As the Size of the project grows, it becomes difficult to configure, test or change the run time behavior of the project without proper interactive interface
- We will integrate CLI interface to our project to make our life easy, We can add any custom show , config CLIs
- We will use CLI library and integrate it with our project
  - Use it with several other C/C++ projects freely
- Appendix C1 and C2 contains a mini-course to walk you through the CLI library we will going to use
- Many of my courses already uses this library to provide CLI interface
- Pls go through appendix C , from next lecture video we will do integration of CLI library with our project
- Skip this entire section if you are already using some other CLI library, pls use with which you are already familiar with

Implementing Complex TCP Servers → Integrating CLI Interface

- config tcp-server <name>
- config tcp-server <name> start
- config tcp-server <name> <ip-addr> <port-no>
- config tcp-server <name> abort
- show tcp-server <name>

Implementing Complex TCP Servers → Integrating CLI Interface

Steps

1. Download LibCLI library from github

   git clone https://github.com/sachinites/CommandParser

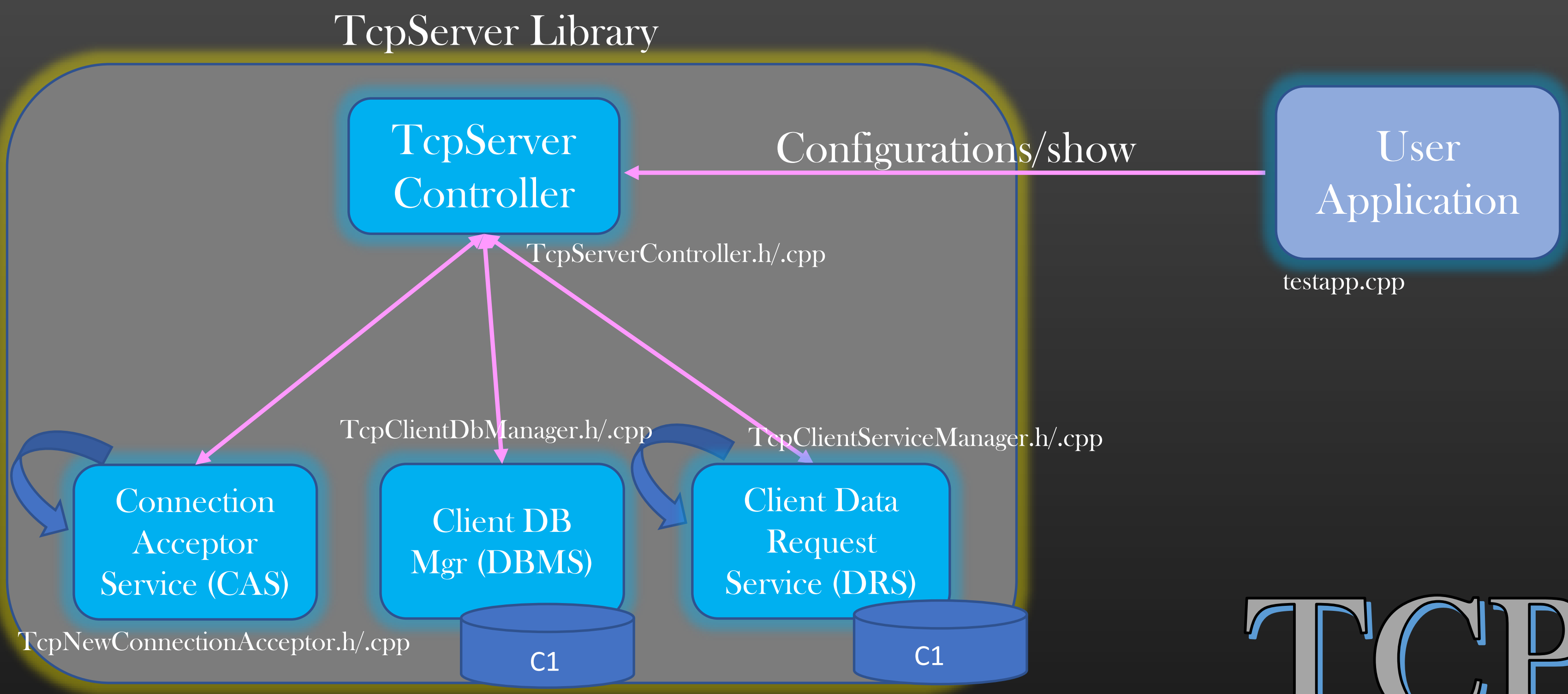
2. Place CommandParser Dir in TCPServerlib/Course

3. Update Makefile now

4. Writing CLIs

   - config tcp-server <name>
   - config tcp-server <name> start
   - config tcp-server <name> <ip-addr> <port-no>
   - config tcp-server <name> abort
   - show tcp-server <name>

Implementing Complex TCP Servers → TCP Server States


TcpServer Library
TcpServer Controller
Configurations/show
User Application
TcpServerController.h/.cpp
testapp.cpp
TcpClientDbManager.h/.cpp
TcpClientServiceManager.h/.cpp
Connection Acceptor Service (CAS)
Client DB Mgr (DBMS)
Client Data Request Service (DRS)
C1
C1
TcpNewConnectionAcceptor.h/.cpp

TCP-Server

States

Implementing Complex TCP Servers → TCP Server States

- If we could keep a track of TCP Server, then it would help us to have better control over the project
- TCP Server States :

```
#define TCP_SERVER_INITIALIZED (1)
#define TCP_SERVER_RUNNING (2)
#define TCP_SERVER_NOT_ACCEPTING_NEW_CONNECTIONS (4)
#define TCP_SERVER_NOT_LISTENING_CLIENTS (8)
#define TCP_SERVER_CREATE_MULTI_THREADED_CLIENT (16)
```

Note :States are not mutually exclusive

CLI : config tcp-server <tcp-server-name> [no] disable-conn-accept

Set flag TCP_SERVER_NOT_ACCEPTING_NEW_CONNECTIONS

TCP Server must Stop the CAS

```
void TcpServerController::StopConnectionsAcceptorSvc();
void TcpServerController::StopConnectionsAcceptorSvc();
```

CLI : config tcp-server <tcp-server-name> [no] disable-client-listen

Set flag TCP_SERVER_NOT_LISTENING_CLIENTS

TCP Server must stop the DRS

```
void TcpServerController::StopClientSvcMgr();
void TcpServerController::StartClientSvcMgr();
```

Implementing Complex TCP Servers → TCP Server States

- Before Proceeeding further, we must first introduce the API in TcpNewConnectionAcceptor class and TcpClientServiceManager class which shall be responsible to start and stop the respective threads

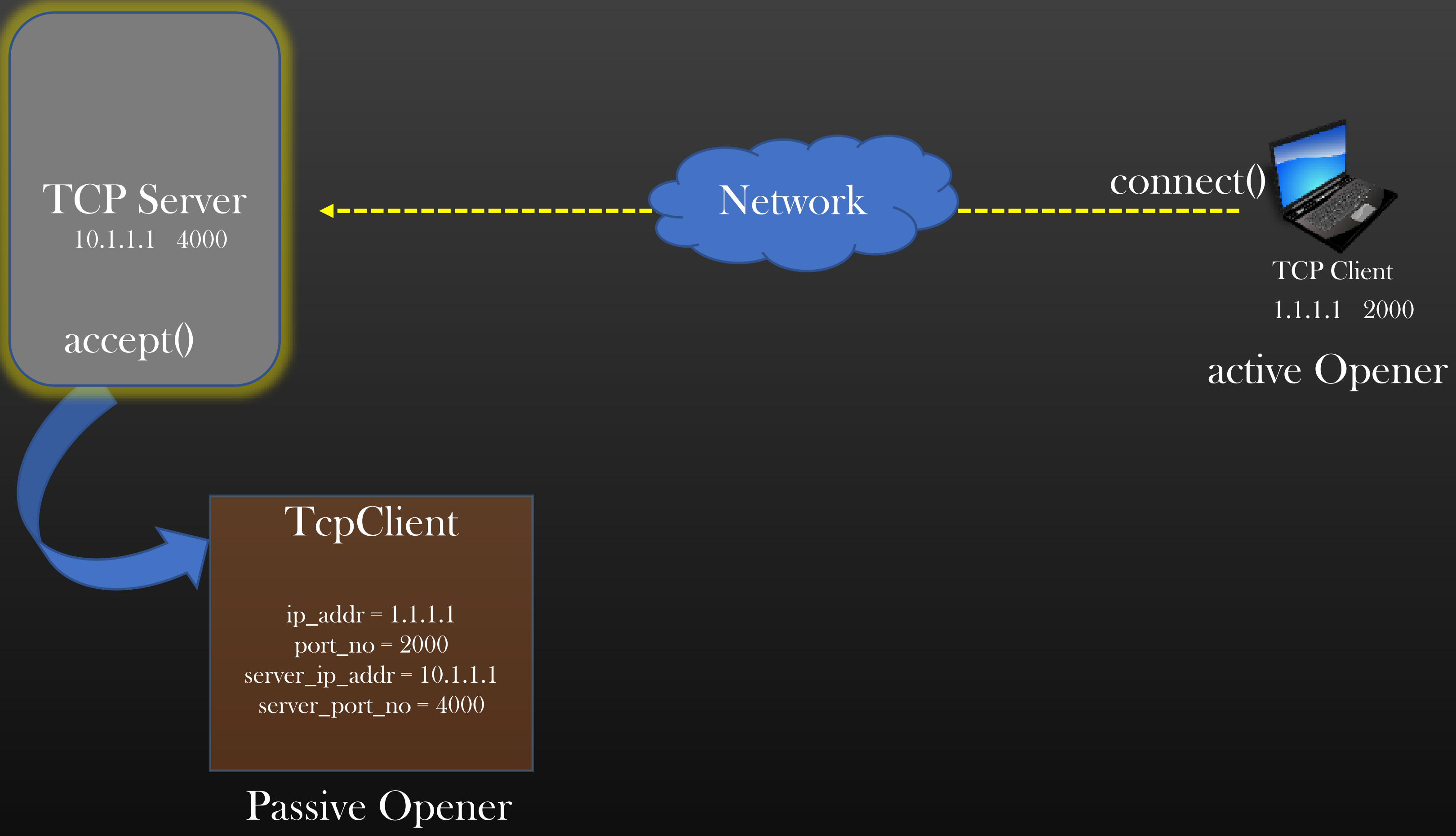
- Let us introduce the Stop() method in both classes

- For Starting the Service threads, we already have APIs (Check Start() of TcpServerController class)

Stop() in CAS class	Stop() in DRS class
Cancel the CAS thread	Cancel the DRS thread
close(this->accept_fd);	Cleanup local client DB
delete the service altogether	delete the Svc altogether

- Finally Implement Stop() method in TcpServerController Class which shall be responsible to shutdown TCPServer, releasing all resources (closing open connections, cancelling all Svc threads, cleaning up all data structures) etc
    - CLI : config tcp-server <server-name> abort

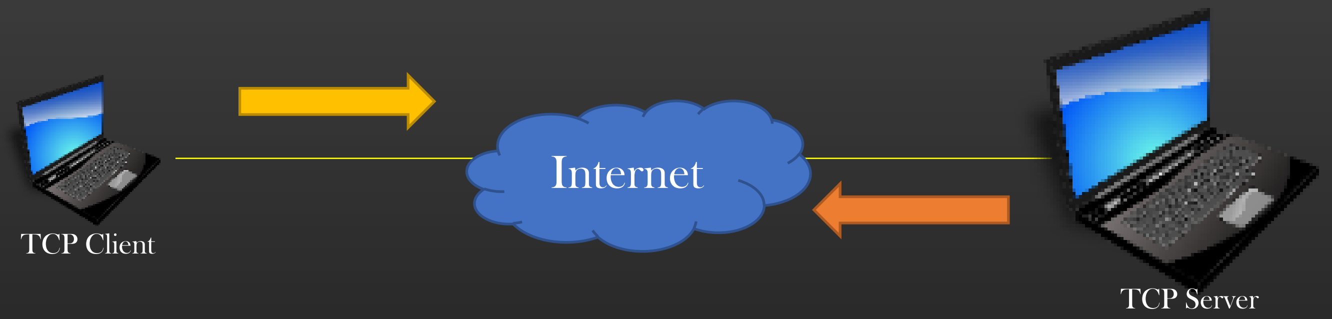
Implementing Complex TCP Servers → Integrating CLI Interface



TCP-Server Client Connection Termination

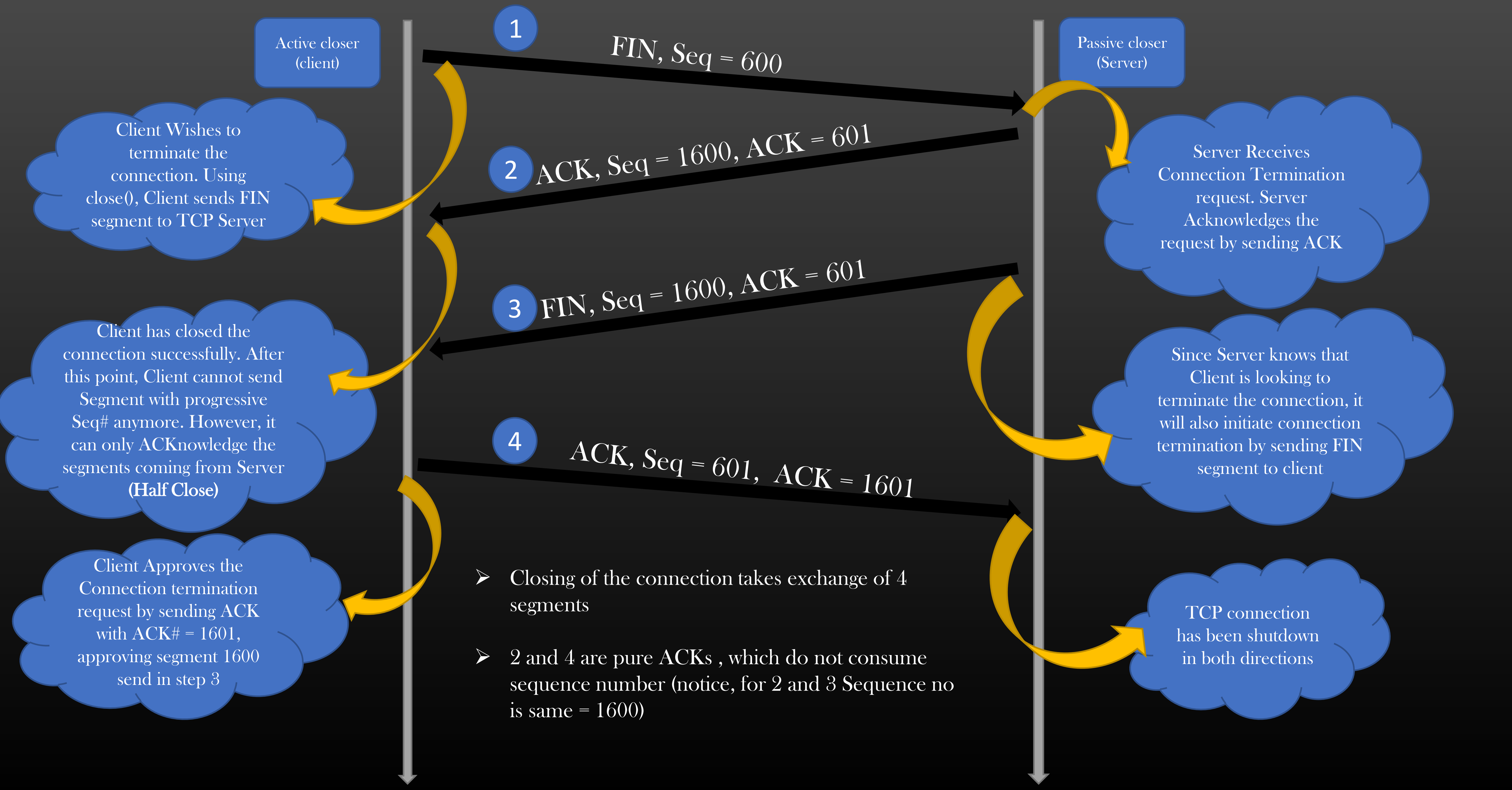
Implementing Complex TCP Servers → Closing Connection

- When either party (Server Or Client) wants to close the connection, they system call close() is used



- Anyone, Client or Server can initiate connection termination by invoking close() on a connection
- Whoever invoke close() first is called active closer, the other one is called passive closer
- Procedure in connection closing . . .

Implementing Complex TCP Servers → Closing Connection



Active closer (client)

Passive closer (Server)

1 FIN, Seq = 600

2 ACK, Seq = 1600, ACK = 601

3 FIN, Seq = 1600, ACK = 601

4 ACK, Seq = 601, ACK = 1601

Client Wishes to terminate the connection. Using close(), Client sends FIN segment to TCP Server

Server Receives Connection Termination request. Server Acknowledges the request by sending ACK

Since Server knows that Client is looking to terminate the connection, it will also initiate connection termination by sending FIN segment to client

Client has closed the connection successfully. After this point, Client cannot send Segment with progressive Seq# anymore. However, it can only ACKnowledge the segments coming from Server **(Half Close)**

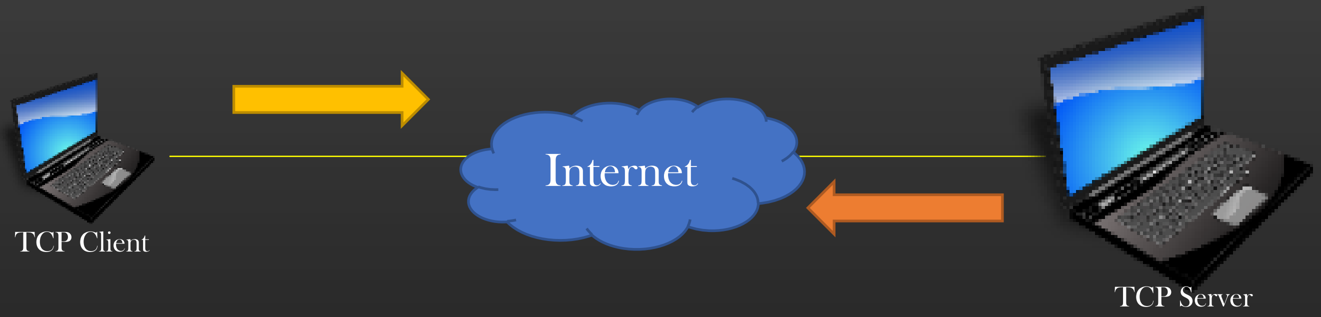
Client Approves the Connection termination request by sending ACK with ACK# = 1601, approving segment 1600 send in step 3

TCP connection has been shutdown in both directions

- Closing of the connection takes exchange of 4 segments
- 2 and 4 are pure ACKs , which do not consume sequence number (notice, for 2 and 3 Sequence no is same = 1600)

Implementing Complex TCP Servers → Closing Connection

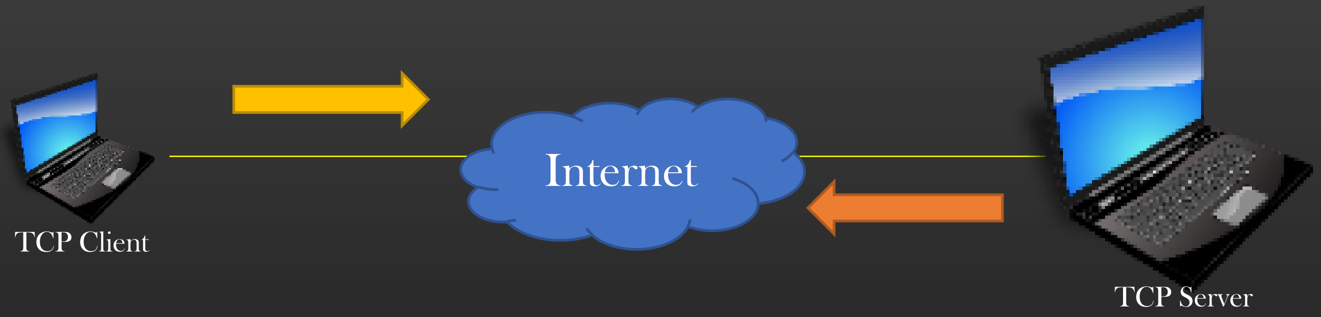
- But I wish things were simpler in real life …



- A FIN pkt may get lost, OR its ACK may get lost
- The network in the middle may have failed
- The Peer Machine may have got crashed
- So, invoking close() doesn't really guarantee that both machine would terminate the TCP connection gracefully
- Hence - Concept of TCP-Keep-Alive messages

Implementing Complex TCP Servers → Closing Connection →Keep Alive Messages

- In the scenario where it is necessary for communicating peer to know that other peer is ALIVE or not, both machines need to periodically exchange TCP Keep-Alive Messages (Heartbeat Messages)



- Let's say both Machines exchanges TCP KA msgs over TCP connection with a periodic time interval of 10s
- Hold time is 15 sec

- Each Peer may either terminate the connection voluntarily by invoking close() Or
- If a machine do not RECV KA msg for hold-time sec, then machine assumes remote peer is no more alive, and hence invoke close() and cleanup the connection

- TCP Specification doesn't say anything about KA msgs. So, it is application's choice to decide exchange of TCP KA alive msgs is required or not . Eg : File Downloader do not need to setup KA msg exchanges.

- You can choose whatever msg format for KA msgs, it differ from application to application. Standard Application standardizes the KA msg format. Eg BGP

Implementing Complex TCP Servers → Closing Connection →Keep Alive Messages

- Let's Enhance our TCPServer
- Our TCP Server would run the Expiration timer per client as soon as a client gets connected, duration of 10 sec
- Any client connected to our TCPServer need to send KA msgs periodically at an interval of 10 sec
- TCP-Server shall refresh the expiration timer as soon as KA msg is recvd from Client
- Our TCP Server abort the client connection if KA msg is not recvd within hold time (15 sec)
- Need to Use Timer Library for this functionality to implement
  - Let's spend 30 minutes to ramp up on using Timer Library
  - If you have your own library, you may use that . . .

Implementing Complex TCP Servers → Closing Connection →Summary

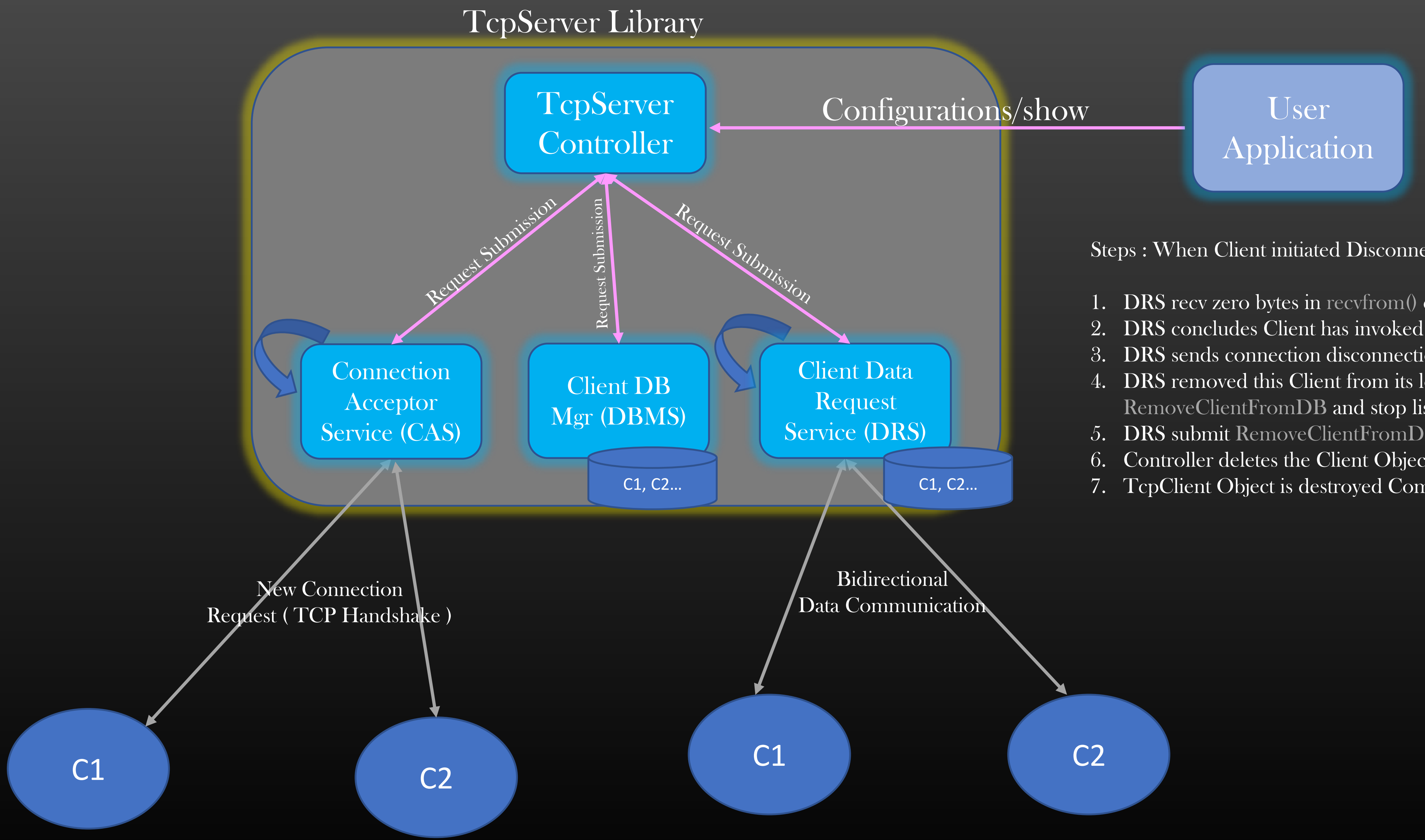
- Communication Parties closes the connection :
  - Either voluntarily by invoking close () when connection is no more required
  - Passively, upon expiration of KA timer

- KA timer ensures that there are no bogus/false connection left open either on client or server side

Implementing Complex TCP Servers → Closing Connection → Implementation

- Let TCP Server has an instance of global timer thread , called Wheel Timer

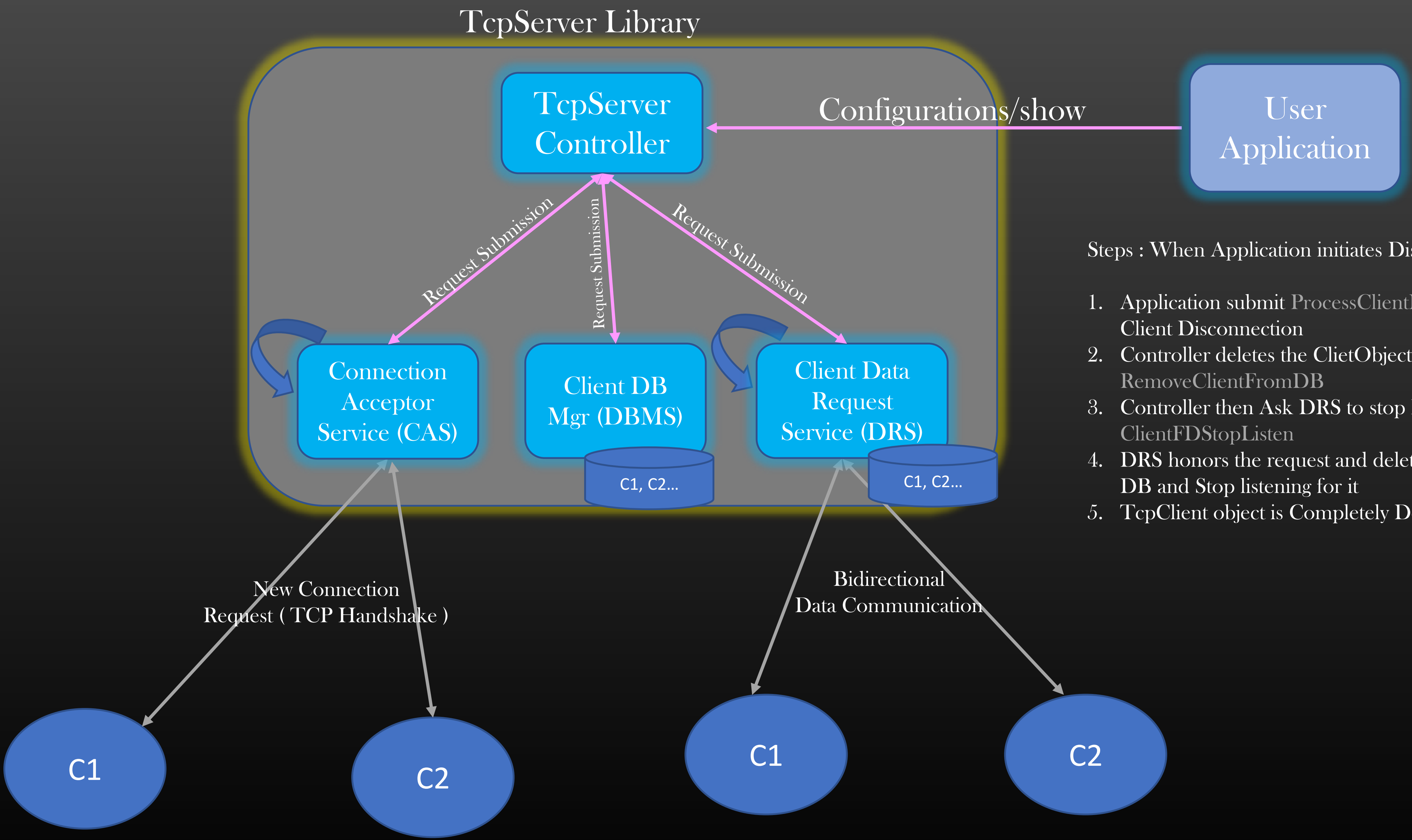
Implementing Complex TCP Servers → Client Originated Dis-Connection Sequence Steps



Steps : When Client initiated Disconnect

1. DRS recv zero bytes in recvfrom() call
2. DRS concludes Client has invoked connection close
3. DRS sends connection disconnection notification to application
4. DRS removed this Client from its local DB using RemoveClientFromDB and stop listening for it
5. DRS submit RemoveClientFromDB Request to Controller
6. Controller deletes the Client Object from Centralized DBMS
7. TcpClient Object is destroyed Completely using Abort()

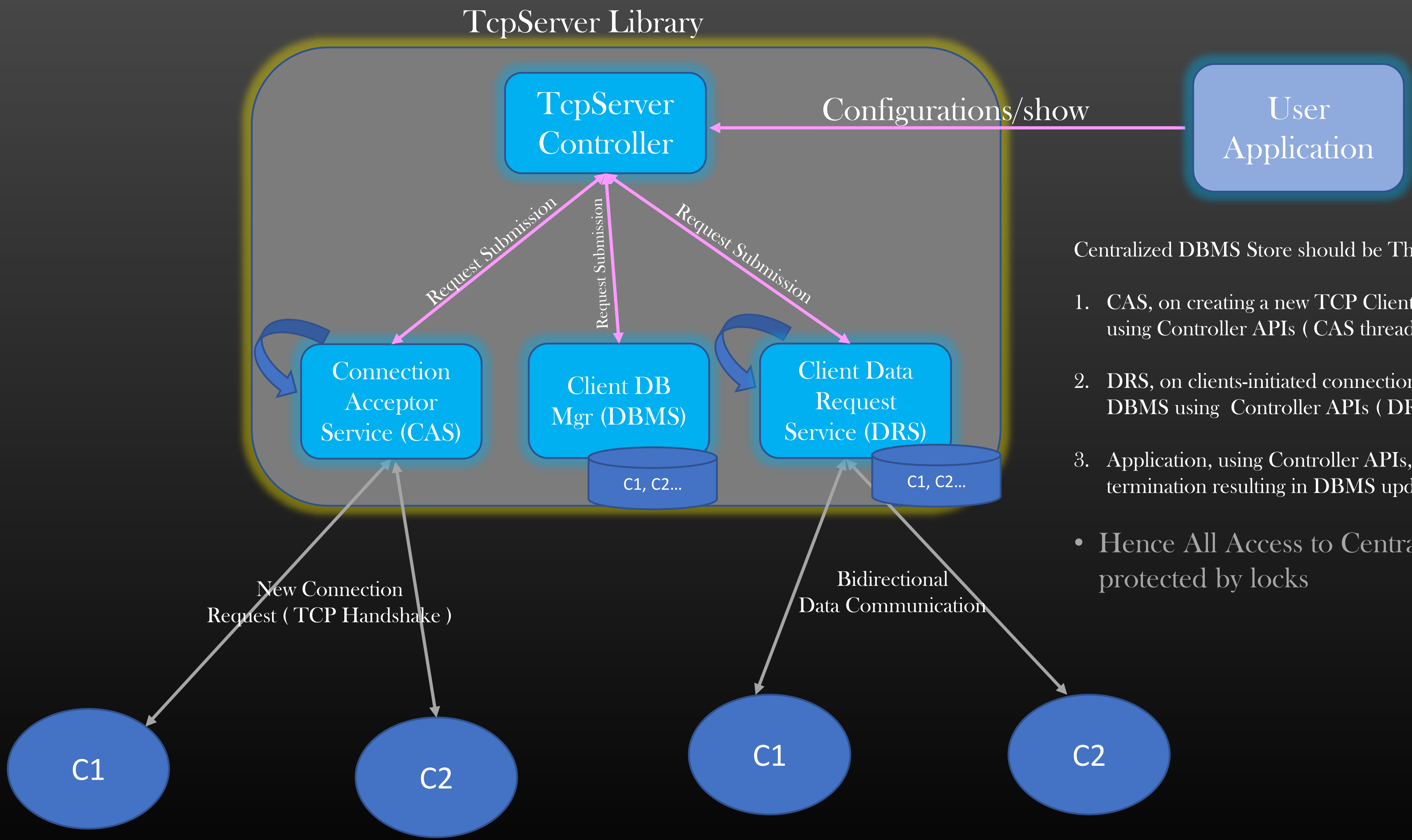
Implementing Complex TCP Servers → Application Originated DisConnection Sequence Steps



Steps : When Application initiates Disconnect for some Client

1. Application submit ProcessClientDelete request to controller for Client Disconnection
2. Controller deletes the ClietObject from DBMS using RemoveClientFromDB
3. Controller then Ask DRS to stop listen for this Client using ClientFDStopListen
4. DRS honors the request and delete Client Object from its local DB and Stop listening for it
5. TcpClient object is Completely Destroyed

Implementing Complex TCP Servers → Central DBMS Access



Centralized DBMS Store should be Thread Safe (protected by locks)

1. CAS, on creating a new TCP Client Object, update the DBMS using Controller APIs (CAS thread in Action)

2. DRS, on clients-initiated connection disconnection, update the DBMS using Controller APIs (DRS thread in Action)

3. Application, using Controller APIs, initiate Client connection termination resulting in DBMS update (Appn thread in Action)

- Hence All Access to Central DBMS by be protected by locks

Implementing Complex TCP Servers → Our Project Design Overview

- So , this was our project high level discussion of base design and features

- We will discuss some more add on later once we finish the project to this point
  - Message liveness detection using Keep-Alives
  - TCPServer in Client Mode
  - Launching Multi-Threaded Client
  - Client Migration

- Let us start with the project implementation

- Will be showing all codes on C like C++ only (no Complex OOPs, No Templates etc)

- Python, Java, JS developers can also pursue this course, they just have to write line-by-line equivalent code

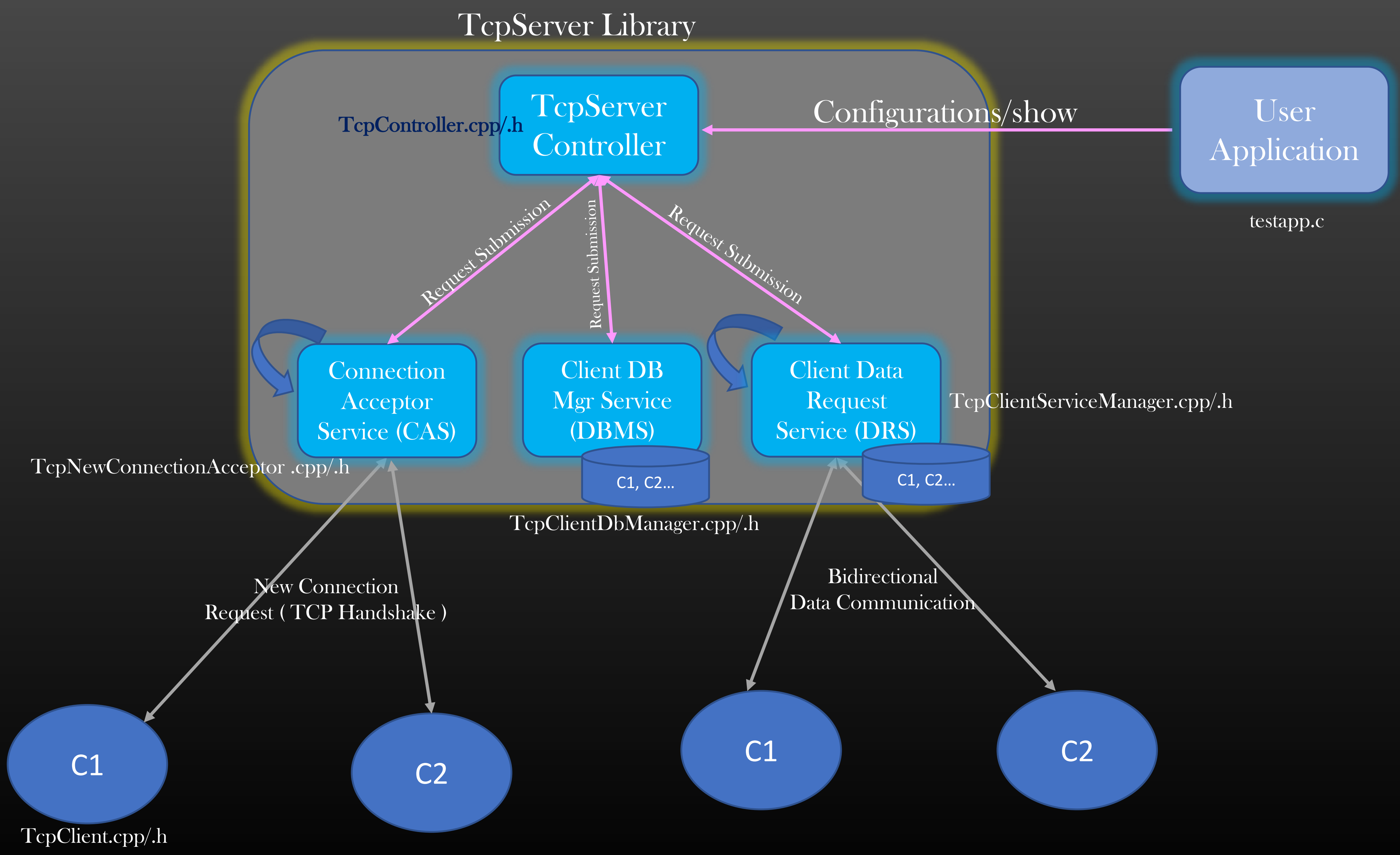
- Thread Lib used : Pthreads
  - You are free to use C++ inbuilt threading lib std::thread instead if you are use to of it
  - Pure C programmers continue to use pthreads only

Socket Programming Implementing Complex TCP Servers

By CSEPracticals

Networking Operating Systems Linux System Programming Kernel Network Protocols TCP/IP
Memory Management IPC RPC Multi-threading Socket Programming Asynchronous Programming

Implementing Complex TCP Servers → Project Files


TcpServer Library
TcpServer Controller
TcpController.cpp/.h
Configurations/show
User Application
testapp.c
Request Submission
Request Submission
Request Submission
Connection Acceptor Service (CAS)
Client DB Mgr Service (DBMS)
Client Data Request Service (DRS)
TcpClientServiceManager.cpp/.h
TcpNewConnectionAcceptor .cpp/.h
C1, C2…
C1, C2…
TcpClientDbManager.cpp/.h
New Connection
Request (TCP Handshake)
Bidirectional
Data Communication
C1
C2
C1
C2
TcpClient.cpp/.h