**Computer Systems Technology**

British Columbia Institute of Technology

COMP 8005 - Assignment2- Design

Albert Huang&

Aiyan Ma

Mar 5, 2018

Revision History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No** | **Date** | **Version** | **Revised by** | **Remarks** |
| 1 | 2018-Feb-26 | V0.1 | Aiyan,Ma | The initial draft for Java version design |
| 2 | 2018-Mar-04 | V1.0 | Aiyan Ma | Adding pseudo code |
| 3 | 2018-Mar-05 | V2.0 | Albert, H. | Rebuild two versions design work |

# Table of Contents

[Table of Contents 1](#_Toc411691241)

[Design Work 2](#_Toc637586518)

[1. Python mode 2](#_Toc1205090687)

[1.1 General Client 2](#_Toc1419452783)

[1.1.1 Flowchart 2](#_Toc1176823636)

[1.1.2 Description 3](#_Toc143220586)

[1.1.3 Pseudo Code 4](#_Toc381884100)

[1.2 Select Server 5](#_Toc1766778639)

[1.2.1 Flowchart 5](#_Toc1629075785)

[1.2.2 Description 6](#_Toc540840778)

[1.2.3 Pseudo Code 7](#_Toc1679754067)

[1.3 Epoll Server 8](#_Toc210790640)

[1.3.1 Flowchart 8](#_Toc552944522)

[1.3.2 Description 9](#_Toc2045860629)

[1.3.3 Pseudo Code 10](#_Toc1729687113)

[1.4 Multi-Thread Server 11](#_Toc1857874483)

[1.4.1 Flowchart 11](#_Toc817358114)

[1.4.2 Description 12](#_Toc858270436)

[1.4.3 Pseudo Code 13](#_Toc540830732)

[2. Java mode 14](#_Toc279314537)

[2.1 Java version Client 14](#_Toc1991875304)

[2.1.1 Flowchart 14](#_Toc1318939454)

[2.1.2 Description 14](#_Toc1050136270)

[2.1.3 Pseudo Code 15](#_Toc195056828)

[2.2 Java Select Server 16](#_Toc1099278039)

[2.2.1 Flowchart 16](#_Toc768424846)

[2.2.2 Description 16](#_Toc916163262)

[State Machine 17](#_Toc567540900)

[Date Structure 17](#_Toc681467551)

[2.2.3 Pseudo Code 18](#_Toc571787206)

[2.3 Java Epoll Server 19](#_Toc1308894267)

[2.3.1 Flowchart 19](#_Toc1093158792)

[2.3.2 Description 19](#_Toc1209373725)

[2.3.3 Pseudo Code 19](#_Toc366501306)

[2.4 Java Multi-Thread Server 20](#_Toc365127927)

[2.4.1 Flowchart 20](#_Toc238713713)

[2.4.2 Description 20](#_Toc509721893)

[2.4.3 Pseudo Code 21](#_Toc747012028)

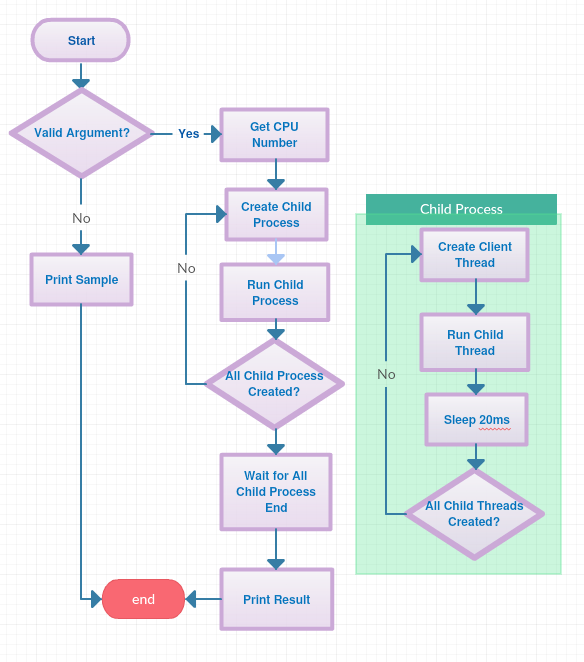
[Appendix 21](#_Toc2005492352)

# Design Work

# Python mode

## 1.1 General Client

### 1.1.1 Flowchart



### 1.1.2 Description

1. Handle user arguments

Allow user input :

* IP address of the server;
* Port of the server;
* How many times it would send and receive data;
* How long time it would wait between 2 sending event;
* How many threads one process will manage to connect the server;

1. Indicate user the pattern of correct input

When user make wrong input, program would print a hint to user

1. Find amount of CPU

Find the amount of CPU and according to this number, generate child process to hold those connections.

1. Create Child process

Create child processes, the amount equal to the number of CPU of this host. During create 2 thread there is a gap, 20ms, to sleep.

1. Create Clients Threads (inside each child process)

Create amount of client thread that connect to server.

1. Client send data to Server (inside child process)

Client create thread that connects to server and sends data to server.

1. Client receives data from server (inside child process)

Client receives echo data from the server.

1. Close Client Process

When all thread finish their work, they would close connection to the server and then close client process

### 1.1.3 Pseudo Code

// Get user settings for Server address, port, repeat times, waiting time and amount of workers under each cpu

// Get amount of cpu cores of the host

// Loop and create child process for amount of cores

// ------ Child Process

// While worker\_created < user\_set\_workers

// create client socket

// send data

// receive data

// increase worker number by child processes

// log results

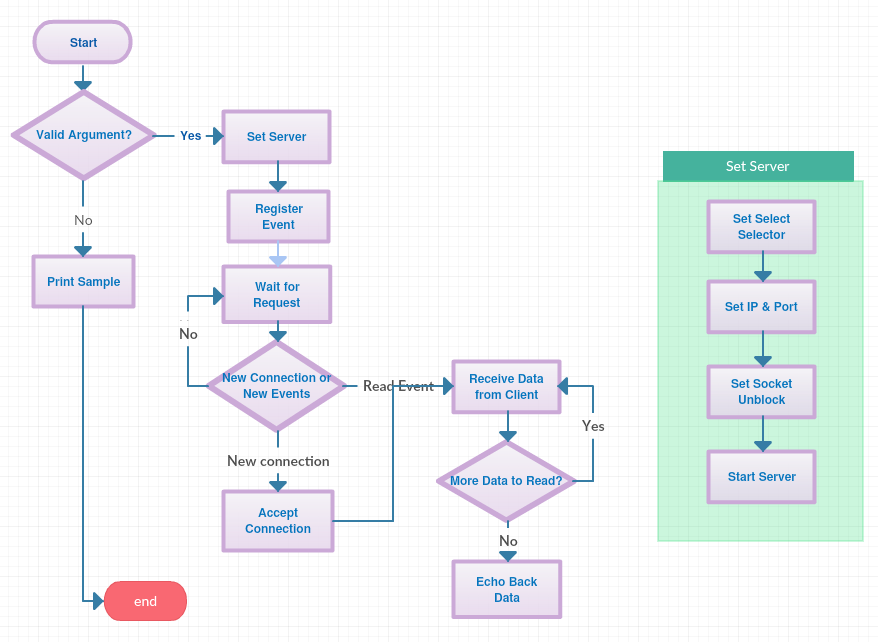
// End While

// Close connections

// -------- End Child Process

## 1.2 Select Server

### 1.2.1 Flowchart



### 

### 1.2.2 Description

1. Handle user arguments

Allow user to select the port to supply service.

1. Indicate user the pattern of correct input

When user make wrong input, program would print a hint to user

1. Setup Server

Select which mode of selector we use select mode or epoll default mode

Create socket to listen on and bind it.

Set socket to unblocking mode.

1. Register event

Server register how to handle what kind of event

1. Accept new connections

Accept all connection request by using infinite loop.

1. Receiving data from client

If the client is connected, it would trigger a read event, the registered function would receive the data.

1. Echo back data to client

If there is no more data to receive, the server would begin send back those data it received moment ago.

### 1.2.3 Pseudo Code

// Set user specified port

// Setup mode to select

// Setup server socket

// While TRUE

// accept connections request

// Foreach request

// if connection is new to server

// add client connection

// else if request is from client

// receive data from client

// if all data received

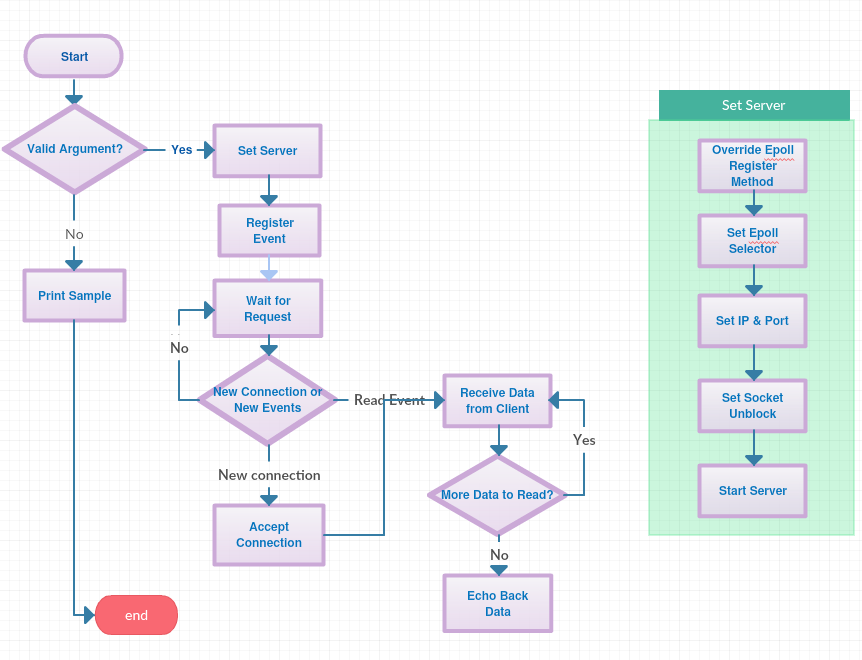
// echo data to client

// else

// continue receiving data

## 1.3 Epoll Server

### 1.3.1 Flowchart



### 1.3.2 Description

1. Handle user arguments

Allow user to select the port to supply service.

1. Indicate user the pattern of correct input

When user make wrong input, program would print a hint to user

1. Setup Server

Select which mode of selector we use epoll edge level mode

Create socket to listen on and bind it.

Set socket to unblocking mode.

1. Register event

Server register how to handle what kind of event

1. Accept new connections

Accept all connection request by using infinite loop.

1. Receiving data from client

If the client is connected, it would trigger a read event, the registered function would receive the data.

1. Echo back data to client

If there is no more data to receive, the server would begin send back those data it received moment ago.

### 1.3.3 Pseudo Code

// Set user specified port

// Setup mode to epoll

// Setup server socket

// While TRUE

// accept connections request

// Foreach request

// if connection is new to server

// add client connection

// else if request is from client

// receive data from client

// if all data received

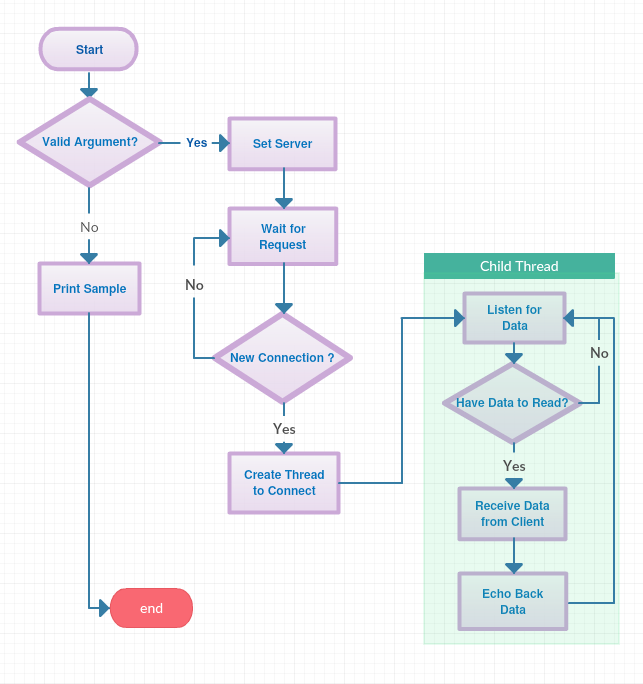
// echo data to client

// else

// continue receiving data

## 1.4 Multi-Thread Server

### 1.4.1 Flowchart



### 1.4.2 Description

1. Handle user arguments

Allow user to select the port to supply service.

1. Indicate user the pattern of correct input

When user make wrong input, program would print a hint to user

1. Setup Server

Create socket to listen on and bind it.

Create thread to handle connection.

1. Accept new connections

Accept all connection request by using infinite loop.

1. Receiving data from client

If the client is connected, it would trigger a read event, the registered function would receive the data.

1. Echo back data to client

If there is no more data to receive, the server would begin send back those data it received moment ago.

### 1.4.3 Pseudo Code

// Set user specified port

// Create socket and bind

// Wait for connection from clients

// Create connection by using thread

// ---- inside worker thread

// listen to client

// if data recv and not timeout

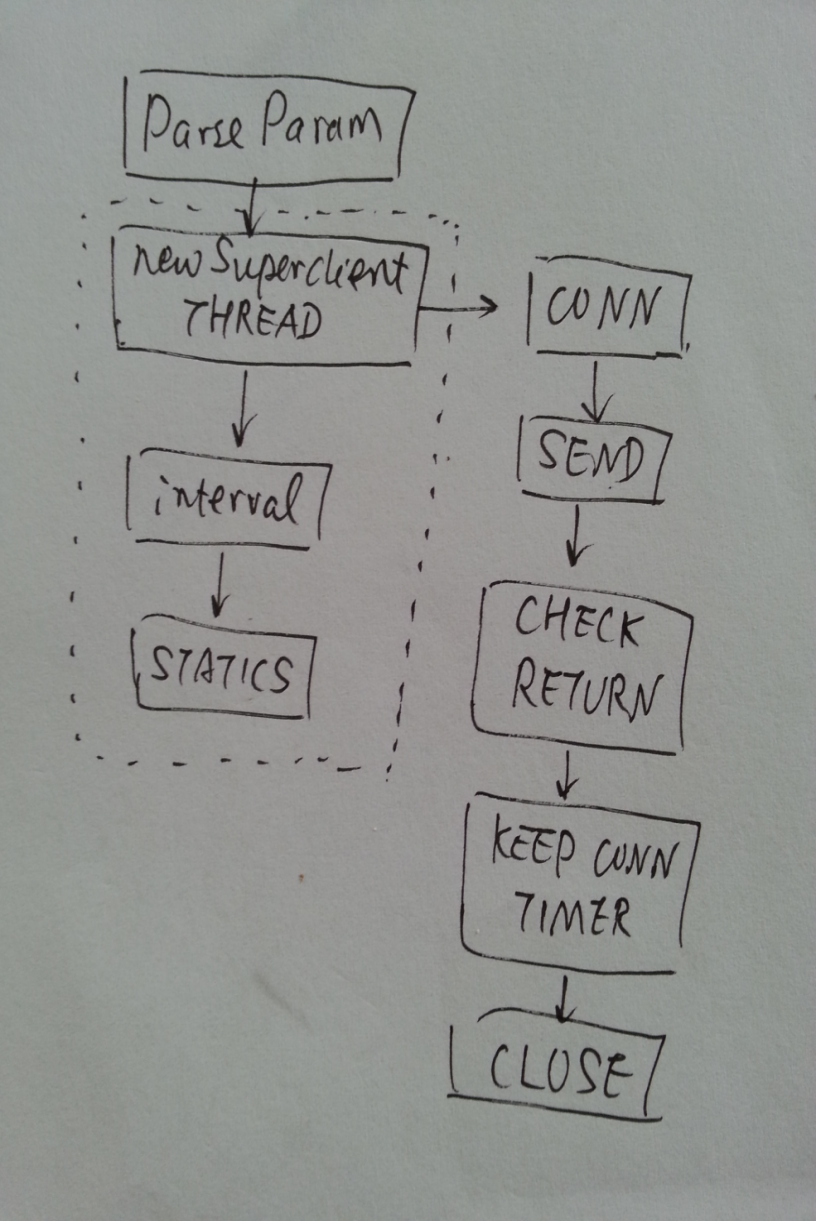
// echo data back

// keep listening to client

# 2. Java mode

## 2.1 Java version Client

### 2.1.1 Flowchart



### 2.1.2 Description

The client working as the below way:

After Parse params for instance buffer\_size, interval, TTL, MESG\_LOOP( for messages sent in one connection), it will continuously create threads to send packets and check the return.

Per the testing on the LINUX, the MAX number of connection can be created is 65530( limited by the ipv4-port-range of the linux kernal).

Note: the content in the dotline square means a loop running…

A super client is designed to challenge the server both from scalability and performance prospective.

It’s based on java.nio and mutil-thread mode, to:

1. Establish connections up to 1000 per second at one client, and how long the connection can last;
2. Support multiple client in one machine, depending on the
3. Construct various size of packets ( from 1 byte to 1024M byte)
4. Send multiple packets in more connection, to stree the server performance
5. Log the messages, connections time used etc.

### 2.1.3 Pseudo Code

\\Initialize parameters(ip, port, buffer\_size, ServerSocketChannel etc)

\\Start the forever loop to do the following:

\\ Create thread of Superclient to do the following:

\\ Connection()

\\ Send() to send the messages

\\ Check return from Server

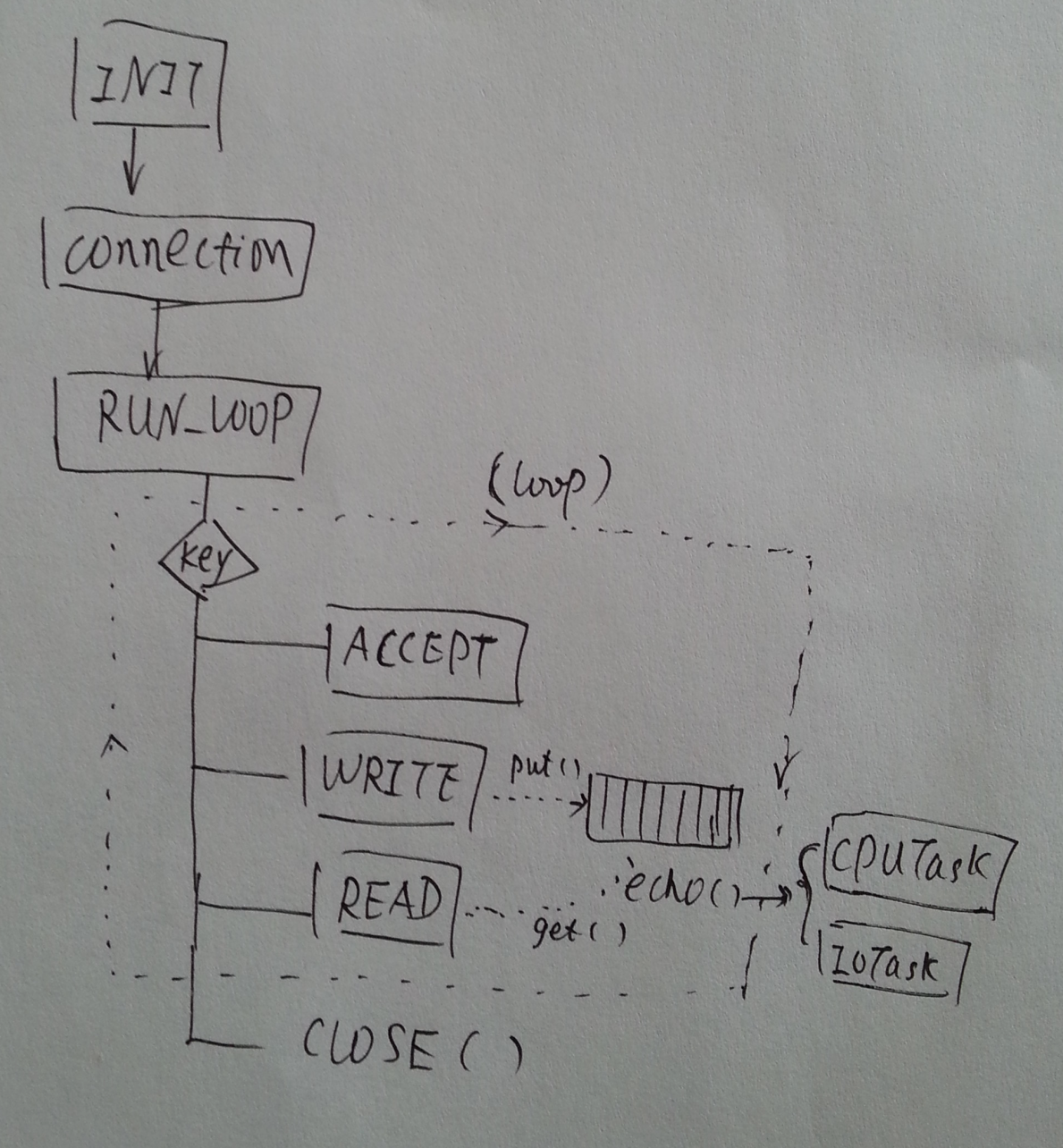
\\ Send more messages if needed

\\ Sleep intervals

\\ Create another superclient thread

## 2.2 Java Select Server

### 2.2.1 Flowchart



### 2.2.2 Description

EPOLL and POLL are pretty much same from application server design point of view, the only different is the Selector Provider underline.

To archive the high performance(fast) and scalability(connection), following method are designed at EPoll Server:

1. EPoll , or edge trigger technology

Through java nio and the SelectorPorviders by different service provider, the server has the ability to work in a edge trigger way.

2) Mesg cache

There is a hashmap designed in the HP-Server to store the key-value of the channel and the data, which can provide the ability on “ fast-cache”;

1. State machie

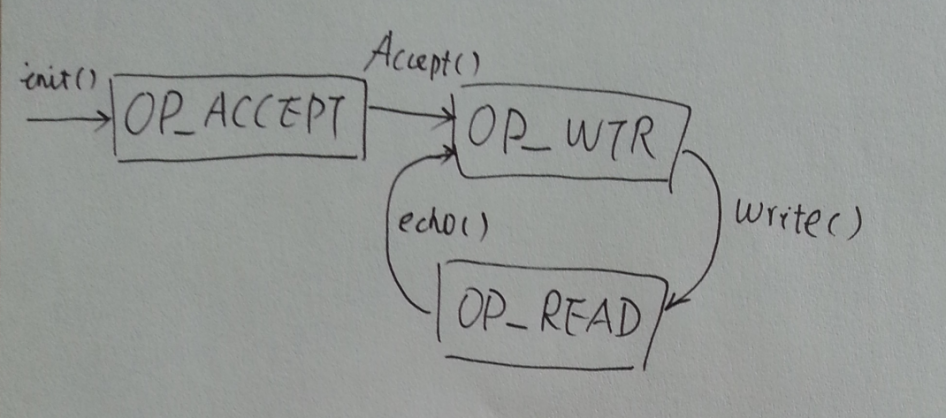
There is on the state machine implemented in the HPServer to maintenance and trigger the transfer of the read and write.

Last need to mention is the ByteBuffer to allocate the fixed size of byte when receiving and sending out the payload, if compared with the “stream” used my MT, it’s a huge improve or revolution to high performance.

After discovered the POLL selectorProvider in Linux JVM, it`s found that it has impact on the application server by raising the concurrency exception, which indicates the POLL provider provided by jvm is really muli-thread in the under layer, thus the hashmap is update to ConcurrentHashMap.

### State Machine

There is a state machine with the main state as OP\_ACCREPT, OP\_WTR ( Writer) and OP\_READ. The transition of each state as below:



### Date Structure

There is a Hashmap ( key, data) in the server to cache the messages

Map<SocketChannel, byte[]> mesgCache = new HashMap<SocketChannel, byte[]>()

After write(), it puts the key and data[] to the mesgCache,

When read(), the mesg is retrieved for sending and removed from the cache.

### 2.2.3 Pseudo Code

\\Initialize parameters(port, buffer\_size, ServerSocketChannel etc)

\\Start the forever loop to do the following:

\\ Iterate SelectedKeys()

\\ For each selected keys(), do:

\\ If (Accept) doAccept()

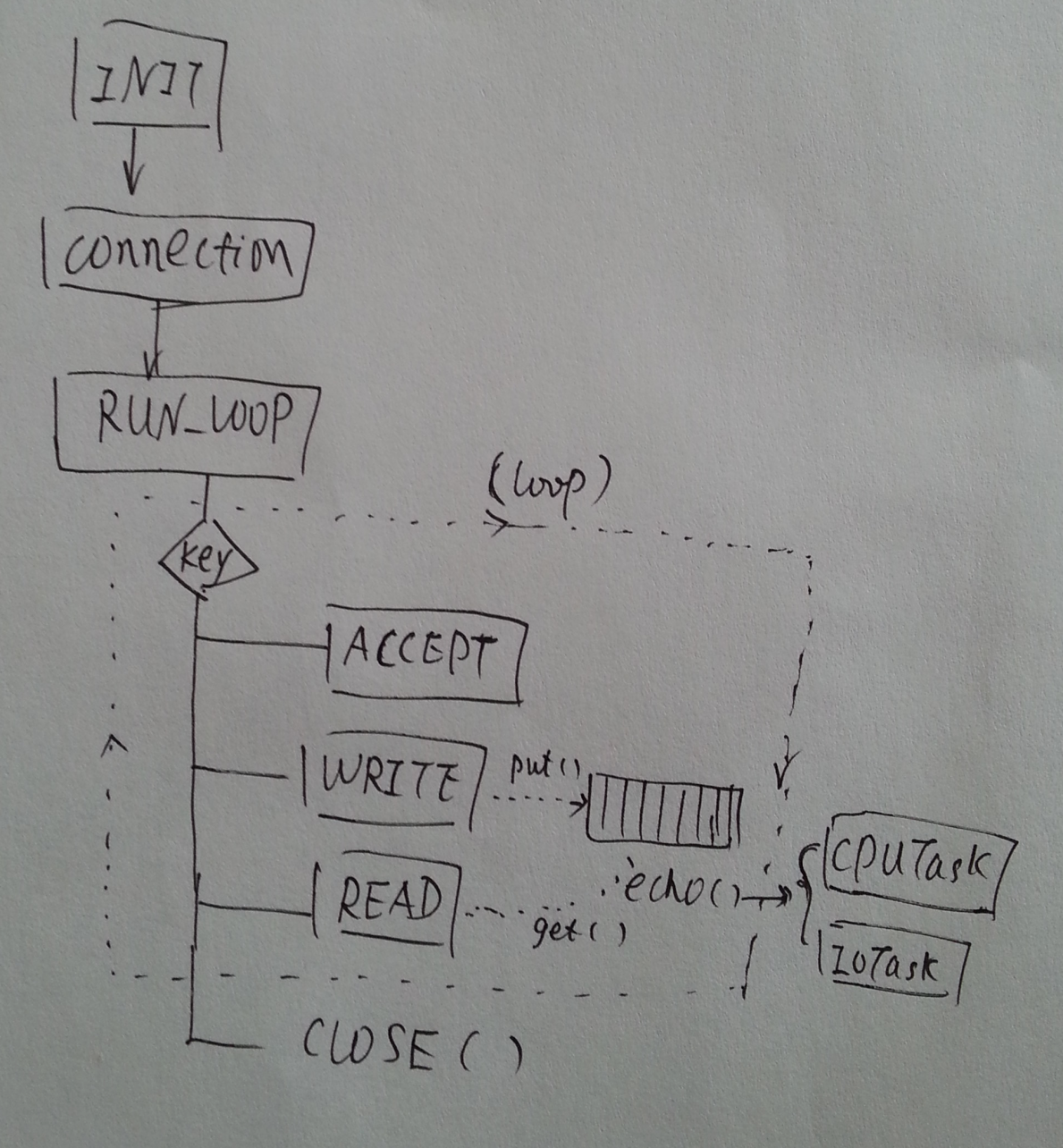
\\ If (Write) doWrite()

\\ If (Read) doRead()

\\Finally, doClose()

## 2.3 Java Epoll Server

### 2.3.1 Flowchart



### 2.3.2 Description

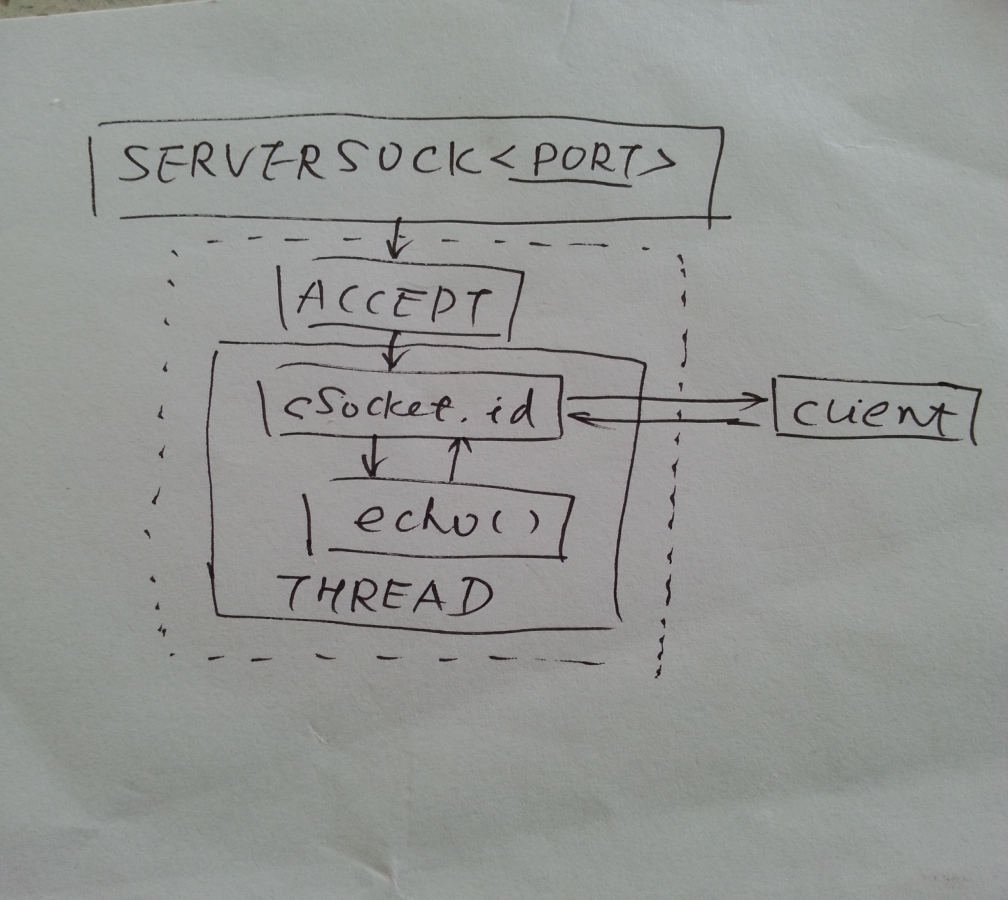
The same as select mode server

### 2.3.3 Pseudo Code

The same as select mode server

## 2.4 Java Multi-Thread Server

### 2.4.1 Flowchart



### 

### 2.4.2 Description

MTServer is using the **one thread per one connection** model, by using InputStream and OutputStream to read and write the data.

When Server Socket starts in the <port> assigned, a forever loop running to create thread with the ClientSocket( cSocket) and the id to do the echo.

### 

### 2.4.3 Pseudo Code

\\Initialize parameters(port, buffer\_size, ServerSocket etc)

\\Start the forever loop to do the following:

\\ Accept to get cSock

\\ Create client Socket with cSocket and thread-id

\\ Echo() to echo back with client request

# Appendix

POLL mode in Linux by Java

By default, in linux, all the nio Selected are built on top of EPOLL, as we can see the

java.nio.channels.spi.SelectorProvider is always set to:

sun.nio.ch.EPollSelectorProvider

And in windows, the SelectorProvider is set to

sun.nio.ch.WindowsSelectorProvider

In Linux, the Poll Selector Provider is:

-Djava.nio.channels.spi.SelectorProvider=sun.nio.ch.PollSelectorProvider.