Project Title: S.M.A.R.T job scheduler

Group members

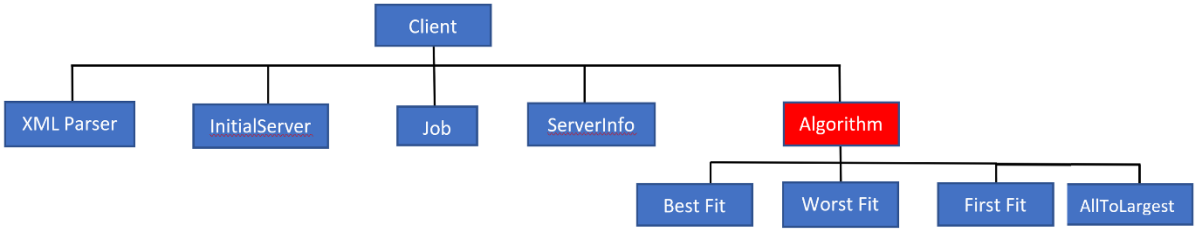
Matthew Chun Hei Lee 45282188

Muhammad Khaled 44978286

Minghou Zou 44208855

Introduction: This is a stage to perform job scheduling procedure to successfully allocate jobs to servers based on memory allocation policies in operating systems. Each memory allocation policy has unique algorithm to select ideal server for each job assignment. In this task, each team member is expected to design and implement accurate algorithm based on individual memory allocation policy, in order to produce identical output generated by defaulted client application.

Design consideration and preliminaries



This stage is based on the client application in previous stage. However, to adapt the interchange of three different algorithms, we have changed the complete structure.

Assumption: there must be at least an available server which has enough **initial** capacity for each job.

XML parser class previously created in Client class, now is extracted as a separate class for better code structure. Available server types, extracted from system.xml using XML parser, are now stored as objects of **InitialServer class**. In all three policies, if there is no ideal servers found based on selection criteria, the job has to schedule on the most ideal server based on the initial capacity, therefore, initialServer would be in handy.

ServerInfo class Memory allocation policies select server for each job assignment based on server availability and server capacity, and data of each server is sent from server-side simulator one after another. Therefore, ServerInfo class is class to store the detail for each server and this class has seven attributes-type, id, state, availableTime, cpuCore, memory and diskSpace.

Job Class – is created to store job information for better coding style.

Algorithm interface choice of algorithm depends on the argument (“-a ff”/”-a bf”/”-a wf”) inserted along with command to run client application. The solution to this issue is to use strategy pattern in java OOP. Algorithm interface has two method signatures, getServer(ServerInfo, Job) and bestServer(). All three separate Algorithm classes- FirstFit, BestFit and WorstFit implement the algorithm interface. As server data would be sent one after another, each server data would be passed into the getServer and compare the availability and capacity with other servers whereas bestServer returns the most ideal server after all comparison. With such pattern, every team member can freely implement their own algorithm using different data structure without interfering code of others, as long as every server would be passed into individual object class and return the most ideal server using bestServer.

Client class algorithm is decided based on what argument inserted therefore client application needs to know how to read arguments. An variable “algorithm” is created and initialized to “ff” as default algorithm. I put a for loop to verify if there is “-a” in the String array “args”. If there is “-a” in the array and the next element is either “bf” or “wf”, algorithm want change to other algorithms accordingly. Algorithm would create for each job, and getAlgorithm would achieve this (return algorithm class acorrdingly to the String algorithm). Each new server send as data from server-side simulator would be created as an object of ServerInfo and comparison method is taken place in getServer() which would be explained in algorithm sections below.

AllToLargest class – implement the same algorithm used in stage 1 – schedule all jobs to the servers with largest Type (available on system xml) and ID=0.

Algorithm description

First Fit-**Muhammad Khaled**

```
import java.util.ArrayList;

public class FirstFit implements Algorithm{
  ServerInfo firstServer;
  boolean isFFFound=false;

  ServerInfo initialServer;
  ArrayList<InitialServer> defaultedServers;
  public FirstFit(ArrayList<InitialServer> defaultedServers) {
    firstServer = null;
    this.defaultedServers=defaultedServers;
    initialServer = null;
  }
  @Override
  public void getServer(ServerInfo server, Job job) {
    if(server==null||server.state==4) return;
    // TODO Auto-generated method stub
    if(!isFFFound && isFitted(job, server)) {
      firstServer=server;
      isFFFound=true;

    }
    if((server.state==1||server.state==3)) {
      if(isInitiallyFitted(job, server)) {
        initialServer=server;
```

firstServer is to store the first server found to have enough current resource capacity to run the job.

isFFFound is Boolean to flag if firstServer is found or not.

initialServer is to store the initial server that has enough initial capacity to run the job.

DefaultedServers passed contains all server types and corresponding initial resource capacities.

getServer is a method to determine the first server having enough capacity to run the job. If the server has state other than 4 (unavailable), the server would be saved in firstServer and flag isFFFound as true. If isFFFound is flagged as true, this if-condition will not execute as there is no need to find the second server.

If servers has booting state/inactive state, these servers would be checked if they have enough initial resource to run the job. Assign to initialServer if they have found one.

```
//a boolean to decide whether the job is capable to run on a server
public boolean isFitted(Job job, ServerInfo server) {
        return (job.cpuCore <= server.cpuCore) && (job.disk <= server.diskSpace)
        && (job.memory <= server.memory);
}

public boolean isInitiallyFitted(Job job, ServerInfo server) {
        String type = server.type;
        for (int i = 0; i < defaultedServers.size(); i++) {
                if (type.equals(defaultedServers.get(i).type)) {
                        return (job.cpuCore <= defaultedServers.get(i).cpuCore) &&
(job.disk <= defaultedServers.get(i).disk)
                                        && (job.memory <=
defaultedServers.get(i).memory);
                }
        }
        return false;
}
```

isFitted is to check if a job can be scheduled on a server based on cpu, diskspace and memory.

IsInitiallyFitted is to check if the server with booting/active state has enough capacity to run the job. It searches through the defaultedServers and collect information of initial capacity of the server type.

```
  @Override
  public ServerInfo bestServer() {
    if(firstServer!=null) {
      return firstServer;
    } else {
      return initialServer;
    }
  }
}
```

BestServer is to return the server used to schedule corresponding job.

Best Fit- **Matthew Chun Hei Lee**

```
public class BestFit implements Algorithm{
	//bestFit algorithm
	int bestFit;
	int minAvail;
	ServerInfo server;


	int initialMinAvail;
	ServerInfo initialServer;

	ArrayList<InitialServer> defaultedServers;

	//constructor
	public BestFit (ArrayList<InitialServer> defaultedServers) {
		bestFit = Integer.MAX_VALUE;
		minAvail = Integer.MAX_VALUE;
		server = null;


		initialMinAvail = Integer.MAX_VALUE;
		initialServer = null;

		this.defaultedServers = defaultedServers;
	}
```

DefaultedServers- is an arraylist of all different types of initialServer extracted from system.xml using XML parser. The rationale of passing this arraylist to the algorithm is that when server is not found, jobs need to be scheduled to the best-fitted server based on their initial resource capacity.

BestFit, minAvail, initialMinAvail would be initialised as maximum integer and comparison between servers would depend on these attributes.

Every new server sent from server-side simulator would be passed to the getServer method, and compare the server already recorded in best fit object and that new server, based on the smallest fitting value and least available time. If the newServer is not unavailable (state=4) and server currently has sufficient capacity, newServer would be used to compare. If the fitnessValue of newServer is smaller than the bestFit value or fitnessVal are equal but newServer has smaller available time than minAvail, newServer would be recorded as a currently ideal server for the job.

```
	//a boolean to decide whether the job is capable to run on a server
	public boolean isFitted(Job job, ServerInfo server) {
		return (job.cpuCore<=server.cpuCore) && (job.disk<=server.diskSpace) &&
				(job.memory<=server.memory);
	}

	public boolean isInitiallyFitted(Job job, ServerInfo server) {
		String type = server.type;
		for(int i=0;i<defaultedServers.size();i++) {
			if(type.equals(defaultedServers.get(i).type)) {
				return (job.cpuCore<=defaultedServers.get(i).cpuCore) &&
(job.disk<=defaultedServers.get(i).disk) &&
(job.memory<=defaultedServers.get(i).memory);
			}
		}
		return false;
	}
	//return the ideal Server using bestFit algorithm
	public ServerInfo bestServer() {

		if(server!=null) return server;
		return initialServer;
	}

}
```

BestFit -keep track of the current ideal server (the minimum difference between job cpuCore and server cpuCore). It is always >=0 as the server with lesser cpuCore than job cpuCore would not be used. **minAvail** -keep track of the available time of the current ideal server. **Server** – keeps track of the current ideal server object

Since there can be a situation in which all servers are running jobs and disk/cpu/memory are updated as jobs are currently running on those servers. Therefore, I need to resolve this issue that client can not find a server that currently has enough resource capacity for the job. Therefore, servers with booting/active state would be used to filter the best-fitted server using their initial resource capacity -(information about the initial resource capacity can be found out using XML parsing on system.xml). since cpu is invariant- servers with same server type would have same resource capacity, it is only necessary to keep track of **initialMinAvail** and **initialServer**.

```
@Override
public void getServer(ServerInfo newServer, Job job) {
	if(newServer.state!=4 &&isFitted(job, newServer)) {
		int fitnessVal = newServer.cpuCore-job.cpuCore;

		if(fitnessVal<bestFit||(fitnessVal==bestFit&&newServer.availableT
		ime<minAvail)) {
			bestFit= fitnessVal;
			minAvail = newServer.availableTime;
			server= newServer;

		}
	} else if(newServer.state==1||newServer.state==3)) {
		if(isInitiallyFitted(job, newServer)) {
			if(newServer.availableTime<initialMinAvail) {
				initialMinAvail = newServer.availableTime;
				initialServer= newServer;
			}
		}

	}
}
```

However, if no servers currently has sufficient resource capacity for the job, the first if-statement will never be executed, as a result, **server** would still point to null. Therefore, second if-statement comes into place to resolve this problem. If the newServer, currently with no sufficient resource capacity for job, is in booting or active state, its initial resource capacity, based on which server type it is, would be used to consider whether it is sufficient for the job using another method called – isInitiallyFitted(). If it has less available time with the recorded initialServer, it replace the initialServer.

isFitted is a boolean method used to determine if server has enough resource capacity for the job.

isInitiallyFitted is a boolean method used to determine if the server has enough initial resource capacity for the job.

bestServer is a ServerInfo method used to return what the ideal server is based on the best fit algorithm. If best-fitted server is not found, return the initial best-fitted server.

Worst fit - **Minghou Zou**

Variables

Worstfit, altFit is being initialized to a minimum Integer number and will be used to compare current worstfit value with the fitness value and to find the worst and altfit server. The AltServer, wstServer, IniServer are going to be returned when the corresponding worst, alternative, initial server are being found. The iniList is a sorted map that is implemented to store the initial servers, so that when worst fit alternative fit is not found, we can return the initial fit server. the server capacity which is the CPU core will be the key and the server will be the value. SortedMap has a function LastKey() which returns the key with the highest value which is the server with the highest capacity.

Functions

IsFitted(Job job, ServerInfo server) this function takes in a serverInfo Object and a job Object and check the current job is able to run on the current server. it will inspect the cpu, memory and disk space requirements for the job and then compare it with server specification and then return a Boolean variable as a results which indicates if the current job is able to run on the server.

GetServers(ServerInfo server, Job job) this function takes a serverInfo Object and a Job Object. In this function, it will determine the worst server, alt server and the initial server

The first condition checks if the current server is fitted for the job and put in the server core as the key and server as the value into the sorted map. The elements in the sorted map, is being sorted according to the server core and the last key will have the biggest number.

The second if statement in the function checks the server state and if the server is fitted for the given job then if the conditions is satisfied it will calculates the fitness value which is server.cpuCore – Job.cpuCore

The third if statement in this function checks if the current worst fit value is less than the new fitness value that is being calculated from above and the server must in the state 3 or 2 which is idle state or the active state (immediately available). If the condition satisfies then it updates the current worst fit and set the current server as the worst server

The fourth if statement in the function checks if the above conditions is not being satisfied it will checks if the fitness value is greater than alternative fit value and server state is equal to zero or server available time is -1 and server state equals to 1. These condition which indicates the current server will only be available in a short definite of time (the booting server and the inactive server will be avalible after a short period of time) then I update the altFit Value and the altServer.

The bestFit function returns the worst fit is found (the current worstfit value is not equal to the initial value) and the else if function returns alternative fit if the alternative fit is found (the current altFit value is not equal to the initial value) and the function returns the intial fit if the above conditions fails it will get the value according to the last key which has the highest fitness value.