# Data Center Simulation Project

## Summary

I was not able to complete the Data Center Simulation Project. I have included everything I have been able to get working so far.

I will discuss the design decisions I made, why I made them, and where I have fallen short.

## Design

I wrote my project in C#. This is an unfamiliar language for me, which probably contributed to some of the difficulty I had. I started out with a mostly object-oriented design. I first designed the network: A Data Center has 4 Regions, each of which has 4 Groups, each of which has 4 Access Points. The Access Points each have 40 available parking spaces. Any data transfer must go from a vehicle, to its Access Point, and so on up to the Data Center and back down the path to the other vehicle. Note that I use “space” and “vehicle” interchangeable. This is because we do not care what vehicle is in a certain parking spot. So vehicles don’t have any identifier of their own, they identify themselves by the parking space they occupy.

Veering somewhat away from the object-oriented design, I created a Global Communications table. This table stores the information we need to keep track of communications.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Space Number** | **Communi-cations Partner** | **Working Vehicle** | **Local Backup** | **Random Backup** | **Status** | **Speed** | **Time Left in Shift** |
| 0 | 0 | 0 | 1 | 2535 | 1 | 0 | 2166 |
| 1 | 1 | 0 | 1 | 2535 | 5 | 0 | 2166 |
| 2 | 2 | 36 | 2 | 147 | 5 | 0 | 23766 |
| 3 | 3 | -1 | -1 | -1 | 4 | 0 | 27366 |

The communications partner is the number of the vehicle we are communicating with. If we are not in a transmit or receive state, the partner is set to ourselves. The next 3 columns are the space numbers of the working vehicle, the local backup, and the random backup. In the section above, we can see that Space 0 has a local backup at space one, and a random backup at space 2535. Both backup copies also track the working vehicle and the other backup. Space 1 knows that it is the local backup for a working vehicle in spot 0, and the random backup is in space 2535. Space 2 is a local backup for space 36, and the random backup is in space 147. And space 3 is currently idle. The status field designates what the vehicle is doing: 1 is Processing, 5 is Holding (a backup), and 4 is Idle (waiting for a job or a backup).

The full list of states is:

|  |  |  |
| --- | --- | --- |
| 0 | justArrived | The car has just arrived in the parking lot |
| 1 | processing | The car is running a job |
| 2 | transmitting | The car is transmitting |
| 3 | receiving | The car is receiving |
| 4 | idle | The car is idle and can accept a job or a backup |
| 5 | holding | The car is holding a backup |
| 6 | holdingForReturn | The car is waiting to transfer its backup back to the primary vehicle |
| 7 | unavailable | It is too close to the end of the shift, and the car is unavailable for accepting either jobs or backups. |

In addition, there is a list of transmission types.

|  |  |
| --- | --- |
| **Transmission Type** | **Description** |
| recInputData | Receiving the initial input data for a job |
| recVM | Receiving the VM from a departing vehicle |
| recBackupData | Receiving backup data from the working vehicle |
| recReturnVM | Receiving the VM back from the backup vehicle -- used to transfer the job back to the |
| transFinalData | transmitting the final data back to the data center |
| transVM | transmitting the VM at the end of shift |
| transReturnVM | transmitting the VM back to the new vehicle |
| transBackups | transmitting backups to our backup vehicles |
| transBackupCopy | transmitting backup from a departing backup vehicle to one taking it's place |
| transLocal | transmitting the local backup |
| transRandom | transmitting the random backup |
| none | not receiving or transmitting |

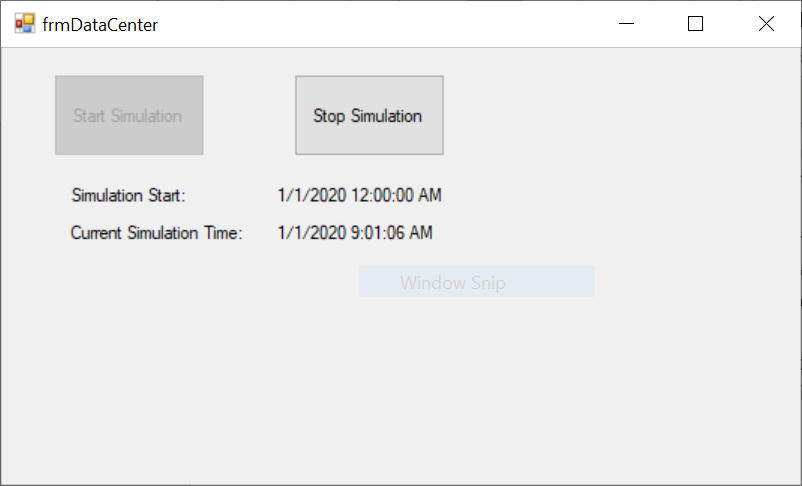
The meat of the program is a time loop. Each pass through the loop represents one second. I chose one second because that is the smallest unit of time which has meaning in the program. Each pass through the look puts a call out to all vehicles to transmit, does some minor administrative work, and then puts out a call to all vehicles to receive. The combination of Status and Transmission type allow the vehicles to communicate with their future selves. For example, if my status is Receiving, and my transmission type is recInputData, I know that when the transmission ends, I next need to create my backups.

## Filling the Parking Lot

At the beginning of the simulation, we have to fill the parking lot. In order to simulate a full parking lot with people parking in random spots, I created an array of numbers, 0 -2559, and then randomized it. Then I looped through the array, assigning the first 320 cars to have 8 hours left in their shift, the next 320 to have 7 hours left and so on down to the last 320 who would only have an hour left. Because the array was randomized, any given cluster would not be divided equally among the shifts. Theoretically, all 4 spots sin a cluster could be on the exact same shift, though that is unlikely. If I were running the data center, I might give employees assigned spots to try to balance the load on any given AP.

I gave each of the cars a job, except for the cars leaving in one hour, which I set to unavailable as they were “too close” to the end of their shift to be able to work on a job and hand it off to a backup. I assigned backups to each of the running cars, but because we were picking up on jobs in progress, I made the decision that the backup data transfers would have occurred prior to the start of the simulation. As a result, the first jobs each car ran would have a faster average completion job than any other jobs picked up later in the simulation.

The initialization happens on Load. Afterwards, the user will be presented with the following window:



You simply click Start to start the simulation, and stop to stop or pause it. If you stop the simulation, you may restart it again from where it left off. You will have to restart the program to get a fresh simulation. Warning – if you pause the simulation to look at the output data, you **must** close the data file before resuming the simulation.

## The Life Cycle of a Vehicle

As I said, the project doesn’t care what vehicle is parked in which spot. So at shift change, each empty space – that is, a space with 0 time left in the shift – is re-initialized. The time left in shift is reset to 8 hours, the current job is null, and there are no backups or communication partners. The status of the car at this point is “justArrived”. The car is then asked to accept a job.

In the AcceptJob function, the first thing that happens for a car with the justArrived status is that we check the global communications table to see if there is anyone who is waiting to return a job to this car space. If there is, then we set our status to recReturnVM and wait for the transfer. For its part, the backup car has been waiting for a shift change so that it can send the VM back. The reason for the decision to send the VM back to the original space was twofold: First, it assured that the job would be going to a car with a full 8 hours left in the shift. The thought was that this would give priority to older jobs. It would also limit the number of times that any given job was bounced around. That would theoretically reduce the total transfer time, especially for longer jobs.

Once the VM is returned, the backup vehicle goes back into a holding pattern, and the new vehicle begins processing where the old one had left off.

If no existing job is found, then the car tries to start a new job. First it will query the data center to see if there are local and random backup cars available. Note that I decided that a car would either be working, be holding a local backup, or be holding a random backup. If I had gotten the project working well, I might have tried to modify it so that a car could hold multiple backups, but since I never did, I was not able to address this issue.

If the car cannot find available backup locations, it sets its status to idle. This makes it available to be used as a backup vehicle for another car.

If the car gets backup locations, then it generates a new job, and sets it’s status to transBackups. It will transmit the input data and job header information to each of the two backups and then begin processing the job.

“Processing” consists of checking each cycle to see if the job has been completed. If it has, the car sets its status to transmitting finalData, and begins to transmit the final data back to the DC. Once the data has been transmitted, the car checks to see if it has time left to start a new job. If it does, it sets its status to idle, and waits to either get a new job or become a backup vehicle.

If the job has not been completed, then the car checks to see if it needs to begin migration. If so, it sets its status to transVM, and notifies the random backup vehicle to be ready to receive. It always chooses the random backup vehicle because if it chooses the local vehicle, that takes 2 slots on the local AP. In order to ensure that the backup vehicle can accept the transfer, random backup vehicles are always set to have at least a 4 hour difference in end of shift from the working vehicle.

After the migration is completed, the car sets its status to unavailable, and waits for its shift to end.

The other option is that the car was asked to be a backup vehicle. Each backup vehicle knows the primary, local, and random spaces for the current job. Each has a copy of the job and a copy of the input data. A backup vehicle, once assigned, will generally remain idle, though with a status of “holding,” while the program is being run. The exceptions are:

* When first designated a backup, the backup vehicle will receive the data from the working vehicle.
* If a backup vehicle notices that it will be leaving soon, it finds a new backup vehicle to transfer its data to. It handles the data transfer itself, leaving the primary vehicle to continue executing the job. After the transfer is complete, the working vehicle is notified of the new backup location.
* If a working vehicle has to leave, it will transfer its VM to the random backup location. It also transmits the intermediate data at the same time. The backup will hold onto the VM until the top of the hour when a new car is occupying the spot, and then it will transfer the VM and intermediate data back.

## Simulation Run

The basic flow of the program is this:

1. At load time, the parking lot is filled with cars with jobs in progress, each with two designated backups which have already received the backup data.
2. Once the user presses the Start button, a one second, potentially infinite loop is started. The loop can be interrupted by pressing the Stop button. A simulation can be started and stopped many times, but it will always pick up where it left off.
3. The structure of the cycle loop is:
   * Call Transmit to gather all transmission requests
   * Middle administrative section – in this section, transmission speeds are determined and other administrative tasks, such as calling for a shift change occur.
   * Call Receive. In Receive, all the transmissions for the current cycle are completed, and vehicles change statuses if necessary. The bulk of the work takes place in the vehicles’ Receive function, so I have included that code at the end of the paper.
4. During the receive part of the loop, any finished jobs will write out their data to a file called ByrneSimOutput.csv which can be found in the bin directory when you are running from within Visual Studio.

Here is a snippet from the output file.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| jobNumber | StartTime | EndTime | JobLength | ElapsedTime |
| 114 | 1/1/2020 0:00 | 1/1/2020 3:11 | 11011 | 11462 |
| 271 | 1/1/2020 0:00 | 1/1/2020 3:11 | 11054 | 11514 |
| 86 | 1/1/2020 0:00 | 1/1/2020 3:15 | 11677 | 11720 |
| 339 | 1/1/2020 0:00 | 1/1/2020 3:17 | 11065 | 11871 |

For each completed job, we record the job number, the start and end times, the reported job length (how long it should take to run) and the elapsed time (how long it actually took from start to finish including any data transmissions. The Simulation starts on 1/1/20 at 12:00am. The job length and elapsed time are recorded in seconds. You can see that for the jobs above, the two values are very close. This is because they were the first jobs to finish and therefore did not have to share bandwidth on the routers, and also because we didn’t include the backup transfers that occurred before the simulation start in their time. As time goes on, theoretically the transfer times would get slower, and the jobs would take longer to complete.

# Postmortem, AKA, What Went Wrong:

The difficulty I had with this project is that we are always dealing with a sliver of time and we need to communicate not only with each other, but also with our future selves. In theory, the setting of states and communication through the Global Communications table should have worked, but it ended up being more fragile and complicated than I had imagined. Every time I would find and fix one issue, 3 more would pop up. At 7:13 pm on the due date, the current problem I am having is that vehicles are losing track of their random backup locations. As a result, they are unable to pass their jobs off, and get them back. This means that any job which takes longer than a shift to run, will not complete. I am out of time, energy, and brain cells to complete the project.

I do want you to know that I did not leave this until the last minute. I have been working on this solidly nights and weekends since the first week in November. I even used my entire Thanksgiving holiday, 4 days of vacation from work, to work on this. It seemed like every time I thought I had it planned out, I came up with new complications I had not considered before. I was having trouble with trying to even figure out a pared down version that I could get working and hopefully expand on. And it didn’t help that I was trying to program in an unfamiliar language that I had never used before. I’m very impressed with the students who were able to get this working. I’ve developed more complex systems for work, but with more time, code reviewers, people to bounce ideas off of, testers, and of course, with not having to have another full time job on top of it. I feel very badly that I was unable to finish the project, and am just hoping that I will get enough partial credit to pass.

## Files

The following files have been included in the .zip file:

1. Data Center Simulation Project.docx – this file.
2. DataCenterSimulation.sln – a Visual Studio file. Open this file with Visual Studio in order to see compile and run the code.
3. Folder DataCenterSimulation – Contains all the C# code files as listed below
   1. Program.cs – autogenerated file to launch the program.
   2. DataCenter.cs. – Main program. It launches the window, initializes the program and runs the main time loop.
   3. RegionHub.cs, GroupHub.cs – these files have no real logic of their own and only serve to pass communications back and forth between the AP and the DC.
   4. AccessPoint.cs -- The access point has limited logic to initialize vehicles.
   5. Vehicle.cs – the bulk of the logic of the program happens in this module.
   6. Node.cs – Node serves as a common ancestor for group, region, and access point.
   7. Job.cs – The job object knows how to create itself, and hold information about the job such as how long it takes, how long it has been running, and what its job number and data sizes are.
4. ByrneSimOutput.csv – the limited output file. When you run the program, this file is created wherever you are running from – either in the same directory as the .exe, or if you are running from within Visual Studio, then it will be in the bin\debug directory.

# Appendix A

// The receive cycle is at the end of the time loop. It checks the status of each car and takes appropriate

// action based on the status and the transmission type, if any.

public void Receive()

{

// Decrement the time left in the shift for the current car.

GV.GlobalComms.Rows[SpaceNumber]["Time"] = (int)GV.GlobalComms.Rows[SpaceNumber]["Time"] - 1;

// Check the car's status

switch ((GV.VMStatus)GV.GlobalComms.Rows[SpaceNumber]["Status"])

{

// If the car was processing, then call to run the current job.

case GV.VMStatus.processing:

// If the current job is complete, set the status to transmitting, and the transmission // type to final data.

if (CurrentJob.Run()) // returns true if job is complete

{

GV.GlobalComms.Rows[SpaceNumber]["Status"] = GV.VMStatus.transmitting;

TransType = GV.TransmissionType.transFinalData;

RemainingDatatoTransmit = CurrentJob.outputDataSize;

GV.GlobalComms.Rows[SpaceNumber]["Partner"] = GV.DCAddress;

}

// If the job didn't complete, then check to see if we are leaving soon and need to start

// migrating our VM. If so, we always migrate to the car holding our random backup. That

// car will send the job back to the new car occupying this same spot.

else if ((int)GV.GlobalComms.Rows[SpaceNumber]["Time"] <= StartMigration)

{

GV.GlobalComms.Rows[SpaceNumber]["Status"] = GV.VMStatus.transmitting;

TransType = GV.TransmissionType.transVM;

int randomBackup = Convert.ToInt32(GV.GlobalComms.Rows[SpaceNumber]["RandomBackup"]);

if (randomBackup == -1)

MessageBox.Show("Random Backup not found");

else

{

GV.GlobalComms.Rows[SpaceNumber]["Partner"] = randomBackup;

RemainingDatatoTransmit = GV.VMSize + CurrentJob.outputDataSize;

DataCenter.NotifyPartnerofTransfer((int)GV.GlobalComms.Rows[SpaceNumber]["RandomBackup"], RemainingDatatoTransmit, GV.TransmissionType.recVM, CurrentJob);

}

}

break;

//If we are holding, we first check to see if we need to offload the backup.

//We do that if we have less than an hour in our shift.

case GV.VMStatus.holding:

if ((int)GV.GlobalComms.Rows[SpaceNumber]["Time"] <= GV.tooClose)

{

int newBackupLoc = -1;

Boolean bBackupFound = false;

int PrimaryCar = (int)GV.GlobalComms.Rows[SpaceNumber]["Primary"];

if (Convert.ToInt32(GV.GlobalComms.Rows[SpaceNumber]["RandomBackup"]) == SpaceNumber)

{

bBackupFound = DataCenter.FindRandomBackupLocation(PrimaryCar);

newBackupLoc = (int)GV.GlobalComms.Rows[PrimaryCar]["RandomBackup"];

}

else

{

bBackupFound = DataCenter.FindLocalBackupLocation(PrimaryCar);

newBackupLoc = (int)GV.GlobalComms.Rows[PrimaryCar]["LocalBackup"];

}

if (bBackupFound)

{

int DataSize = CurrentJob.inputDataSize;

DataCenter.NotifyPartnerofTransfer(newBackupLoc, CurrentJob.inputDataSize,

GV.TransmissionType.recBackupData, CurrentJob);

// Let the new backup vehicle know what's up.

DataCenter.AssignJobtoVehicle(CurrentJob, newBackupLoc, PrimaryCar, DataSize);

GV.GlobalComms.Rows[SpaceNumber]["Status"] = GV.VMStatus.transmitting;

TransType = GV.TransmissionType.transBackupCopy;

RemainingDatatoTransmit = DataSize;

}

}

break;

// Holding for return is a special status meaning that the primary car running our job has decided to

// leave and has passed us their VM and intermediate data. We have to hold onto the job until the new

// car arrives at the start of the next shift. When that happens, we change our status to transmitting

// and notify the new car that it will be getting the VM back so that it can resume the job our predecessor

// began.

case GV.VMStatus.holdingForReturn:

// Wait until the top of the next hour to send the VM back to the new car

if (GV.intElapsedTime % GV.oneHour == 0)

{

GV.GlobalComms.Rows[SpaceNumber]["Status"] = GV.VMStatus.transmitting;

TransType = GV.TransmissionType.transReturnVM;

RemainingDatatoTransmit = GV.VMSize + CurrentJob.outputDataSize;

DataCenter.NotifyPartnerofTransfer(Convert.ToInt32(GV.GlobalComms.Rows[SpaceNumber]["Primary"]), RemainingDatatoTransmit,

GV.TransmissionType.recReturnVM, CurrentJob);

}

break;

// If we are idle or unavailable, we do nothing. The unavailable status will be changed when a new car

// arrives in the spot. The idle status will change when we accept a job or a backup.

case GV.VMStatus.idle:

case GV.VMStatus.unavailable:

break;

// There are many transmission types, and each has diffent actions associated with it.

default: // transmitting or receiving

// decrement the remaining data count by the effective speed of transmission.

RemainingDatatoTransmit -= Convert.ToDouble(GV.GlobalComms.Rows[SpaceNumber]["Speed"]);

// If we have finished transmitting:

if (RemainingDatatoTransmit <= 0)

{

switch (TransType)

{

//Done receiving input data, need to make 2 backups.

case GV.TransmissionType.recInputData:

GV.GlobalComms.Rows[SpaceNumber]["Status"] = GV.VMStatus.transmitting;

TransType = GV.TransmissionType.transBackups;

BackupsNeeded = GV.BackupsNeeded.BothBackups;

RemainingDatatoTransmit = GV.inputDataSize;

if ((int)GV.GlobalComms.Rows[SpaceNumber]["LocalBackup"] == -1)

{

MessageBox.Show("Primary Backup Vehicle invalid", "Fatal Error",

MessageBoxButtons.OK, MessageBoxIcon.Exclamation);

}

else

GV.GlobalComms.Rows[SpaceNumber]["Partner"] = GV.GlobalComms.Rows[SpaceNumber]["LocalBackup"];

DataCenter.NotifyPartnerofTransfer((int)GV.GlobalComms.Rows[SpaceNumber]["RandomBackup"], RemainingDatatoTransmit, GV.TransmissionType.recBackupData, CurrentJob);

break;

// Check to see if both backups are complete. If not, start the second backup.

case GV.TransmissionType.transBackups:

if (BackupsNeeded == GV.BackupsNeeded.BothBackups)

{

BackupsNeeded = GV.BackupsNeeded.RandomBackup;

RemainingDatatoTransmit = GV.inputDataSize;

if ((int)GV.GlobalComms.Rows[SpaceNumber]["RandomBackup"] == -1)

{

MessageBox.Show("Secondary Backup Vehicle invalid", "Fatal Error",

MessageBoxButtons.OK, MessageBoxIcon.Exclamation);

}

else

GV.GlobalComms.Rows[SpaceNumber]["Partner"] = Convert.ToInt32(GV.GlobalComms.Rows[SpaceNumber]["RandomBackup"]);

DataCenter.NotifyPartnerofTransfer((int)GV.GlobalComms.Rows[SpaceNumber]["RandomBackup"], RemainingDatatoTransmit, GV.TransmissionType.recBackupData, CurrentJob);

}

else if (BackupsNeeded == GV.BackupsNeeded.LocalBackup || BackupsNeeded == GV.BackupsNeeded.RandomBackup)

{

if ((int)GV.GlobalComms.Rows[SpaceNumber]["Time"] > TimeToTransferFinalData + CurrentJob.jobLength)

StartMigration = 0;

else

StartMigration = TimeToTransferVM;

GV.GlobalComms.Rows[SpaceNumber]["Status"] = GV.VMStatus.processing;

TransType = GV.TransmissionType.none;

BackupsNeeded = GV.BackupsNeeded.none;

}

else // (BackupsNeeded == GV.BackupsNeeded.none

MessageBox.Show("Transmitting backups when none needed.", "Fatal Error",

MessageBoxButtons.OK, MessageBoxIcon.Exclamation);

break;

// Ive offloaded my backup to another vehicle, and am waiting to leave. Set my

// status to unavailable.

case GV.TransmissionType.transBackupCopy:

GV.GlobalComms.Rows[SpaceNumber]["Status"] = GV.VMStatus.unavailable;

break;

// I'm done receiving backup data, set my status to holding

case GV.TransmissionType.recBackupData:

GV.GlobalComms.Rows[SpaceNumber]["Status"] = GV.VMStatus.holding;

TransType = GV.TransmissionType.none;

break;

// I received the backup from a departing vehicle, now I have to hold it until

// the new vehicle arrives.

case GV.TransmissionType.recVM:

if (Convert.ToInt32(GV.GlobalComms.Rows[SpaceNumber]["Primary"]) == -1)

{

//MessageBox.Show("Can't receive VM. Primary is -1");

}

else

{

GV.GlobalComms.Rows[SpaceNumber]["Status"] = GV.VMStatus.holdingForReturn;

TransType = GV.TransmissionType.none;

RemainingDatatoTransmit = GV.VMSize + CurrentJob.outputDataSize;

}

break;

// I am the new car and have received a VM from a backup car. I start

// running the program.

case GV.TransmissionType.recReturnVM:

GV.GlobalComms.Rows[SpaceNumber]["Status"] = GV.VMStatus.processing;

break;

// I am a departing car and have finished transmitting my VM. I become unavailable.

case GV.TransmissionType.transVM:

GV.GlobalComms.Rows[SpaceNumber]["Status"] = GV.VMStatus.unavailable;

break;

// I completed my job! If it's too close to time to leave, I become unavailable.

// Otherwise I set myself idle so that I can be assigned a new job or can become

// a backup for another car.

case GV.TransmissionType.transFinalData:

if ((int)GV.GlobalComms.Rows[SpaceNumber]["Time"] <= GV.tooClose)

{

GV.GlobalComms.Rows[SpaceNumber]["Status"] = GV.VMStatus.unavailable;

}

else

{

GV.GlobalComms.Rows[SpaceNumber]["Status"] = GV.VMStatus.idle;

}

// After the job is complete, write out the data to the .csv file. The

// actual write to file takes place in the data center once per seond for

// all cars. This is to help the speed of the simulation.

string OutputString = CurrentJob.jobNumber + ","

+ CurrentJob.StartTime.ToString() + ","

+ GV.currentTime.ToString() + ","

+ CurrentJob.GetJobLength().ToString() + ","

+ (GV.currentTime - CurrentJob.StartTime).TotalSeconds + ","

+ SpaceNumber;

GV.csv.AppendLine(OutputString);

DataCenter.DeleteBackup((int)GV.GlobalComms.Rows[SpaceNumber]["LocalBackup"], (int)GV.GlobalComms.Rows[SpaceNumber]["RandomBackup"]);

GV.GlobalComms.Rows[SpaceNumber]["LocalBackup"] = -1;

GV.GlobalComms.Rows[SpaceNumber]["RandomBackup"] = -1;

GV.GlobalComms.Rows[SpaceNumber]["Primary"] = -1;

// set stop time

break;

default:

break;

}

break;

} // switch on transmission type.

break;

} // switch on VM status

} // Receive