# Building the Stage for Coalescent Computing

Akhil Kodumuri
*Computer Engineering*
*University of Illinois at Urbana Champaign*
*Champaign, IL, USA*
*Email: akhilvk2@illinois.edu*

Kyle C.Hale
*Illinois Institute of Technology*
*Chicago, IL, USA*
*Email: khale1@iit.edu*

*Abstract*—**Rapid technological advances in wireless and networking technologies are making shared computing a reality. A user will be able to borrow the computational power from a local device and use it for the duration of its task or while maintaining a viable connection. However, before this idea can become a reality, a platform for testing and developing this technology needs to be developed. In order to begin to build a platform to accomplish Coalescent Computing tasks, two distributive platforms were attempted to be configured. Though this project did not succeed in configuring either of these platforms, it outlines the errors that occurred during its configuration and the final errors that ultimately halted the attempt of the build. The goal of this report is to hopefully help a future developer to eventually solve the errors outlined and eventually lead to a successful build of a testbed for Coalescent Computing.**

## 1. Introduction

With the advancement of wireless internet, computational devices will be able to share and communicate information at a faster rate. These advancements in wireless technology will develop the field of distributive systems because computers will be able to communicate and share information at a faster rate. Users of these computers will start to lose the distinction between local and remote resources [1]. This benefits and highlights the field of Coalescent Computing [1]. However, before this concept can become a reality, a platform for testing and conducting Coalescent Computing related tasks must be made. This platform ultimately must be able to allow for ephemeral connections because that is the life cycle of an ideal application of Coalescent Computing. Though this paper will not be going into building these ephemeral connections, the ultimate goal of this report is to lead to the eventual build of an ephemeral platform. This report looks into two possible candidates to help build a platform that can be used to test Coalescent Computing connections. The platforms being discussed provide online documentation, but the documentation is rather confusing and do not lead to a straight forward build.

LegoOS is a distributive operating system designed for re-inventing how servers are managed in datacenters [4].

TABLE 1: Hardware specifications used in LegoOS setup

| Hardware | Vendor |
|---|---|
| CPU | CPU E5-2670 v3 @ 2.30GHz |
| Server | Intel Xeon |
| Instruction Set | x86-64 |
| Infiniband NIC | MT27500 |

LegoOS has a split kernel architecture in which the LegoOS kernel is split into separate components called monitors. Each monitor manages a different part of hardware within the server. Within LegoOS there are three different types of monitors: storage, process, and memory. As their names suggests, the memory monitor manages the memory unit of the server. The Storage monitor manages the storage of the computer (an example would be SSD), and the process monitor manages the CPU of the server. This paper goes over the a process of building a basic configuration of LegoOS by following the directions specified on the LegoOS repository. This basic configuration requires two separate machines: one for the processor monitor and another for the memory monitor. Apart from LegoOS, this paper describes an attempt to build GiantVM [2], a type of virtual machine created by a distributive hypervisor in which two virtual machines are configured to work as one. It should be noted that all the hardware used in these configurations use the Chameleon Cloud testbed [3] provided by the University of Chicago and Argonne National Laboratory.

The rest of this report will be dedicated to providing a guide to configuring LegoOS and GiantVM as well as showcasing some bugs that may occur if this experiment were to be replicated using Chameleon or a similar setup.

## 2. LegoOS Configuration

The process of configuring LegoOS begins by renting out two instances from Chameleon. Table 1 and Table 2 show the machine specifications of the instances checked out from Chameleon.

The two instances, described by Tables 1 and 2, were checked out for the creation of LegoOS. One instance was used to host the memory monitor while the other will be used to host the process monitor. LegoOS allows for

TABLE 2: Software specifications used in LegoOS setup

| Software | Version |
|---|---|
| Operating System | Centos7.2 |
| GCC | 4.8.5 |
| Kernel | 3.11.1 |

configurations on bare-metal and virtual machines. Since this configuration was done on the Chameleon Cloud[3], the only way to access the instances are via SSH. LegoOS is an operating system, thus, many of its configurations will require multiple reboots and configuration on boot. This won't be possible to do because the SSH connection breaks. Chameleon does offer a console on their website, but, its use is often times unavailable. Because of these reasons, it was decided to configure LegoOS with the use of Qemu virtual machines. It should be noted that the separate monitors of LegoOS communicate with each other with the use of Infiniband connected to a common Infiniband switch. The virtual machines used in this configuration had sole ownership and use of the Infiniband on the host's system because they were created by virt-manager and PCI pass through was used during the creation of each virtual machine. To verify that these two virtual machines are indeed able to communicate with each other using Infiniband the following command can be used:

```
Server: ibv_rc_pingpong
Client: ibv_rc_pingpong <vm_ip_address>
```

After these sequence of commands, it will be evident whether or not Infiniband communication is possible with a specific configuration.

## 2.1. Creating Virtual Machine Environment

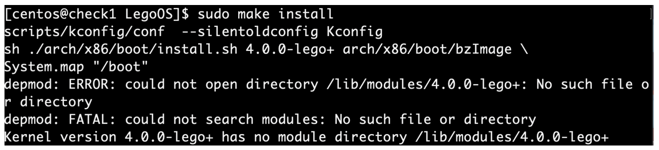
As noted in the beginning of this report, this work was all performed using the resources from the Chameleon Cloud. The resource allocation for this project was shared among other students participating in the REU BigDataX program. There was a shared network configuration for the entire resource allocation. The security settings for the shared network did not allow for dynamic IP allocation for the virtual machines created on the bare-metal hosts. For this reason, it was necessary to create a new private network. Chameleon provides documentation in order for a user to create their own private network. With a private network an IP would be able to be dynamically allocated to the virtual machines. This was a necessary step because, by default, Qemu virtual machines create a default NAT network configuration for the created virtual machine along with a virtual bridge to the host. This will allow the guest virtual machine to communicate with instances within the hosts local area network. This communication, however, is one-sided. Qemu does allow users to change this NAT network to a bridged network, but once created, connection to the host instance is lost and creating a SSH connection to the virtual machine or host times out. It is unknown whether or not this is caused by an incorrect configuration on our part or by the network security configuration on Chameleon. The simplest way to go further with the project was to create a personal private network for this project. These insights are are included in this report because many REU and undergraduate students will use Chameleon in the future and the hope is that this report can provide guidance on maneuvering the Chameleon testbed. After creating a personal network, a router must be created for instances to be able to connect to the personal network. Once the router and network are created, be sure to attach both personal network and the default network created for the shared resource allocation to the instances created. This is because some errors occurred when trying to SSH to the public IP being used to connect to the NIC (on the bare-metal host) associated with the personal network. After launching instances connected to the private network, a bridged network can now be made with the personal network and an IP can be dynamically allocated to the virtual machine.

## 2.2. Installation Process

Once Infiniband connection is confirmed on the virtual machines, LegoOS can properly be compiled and configured. Each step mentioned in this report must be completed on each computer being configured into a LegoOS monitor. First, the .config script of LegoOS must be configured. LegoOS allows for the script to be configured with the typical Kconfig method similar to how generic Linux kernels are compiled. A simple

```
make defconfig
```

is all that is needed to configure the .config script. This script must be generated on both virtual machines because each will become a LegoOS monitor. Once the .config is created, the LegoOS kernel must be compiled. This is just a make command in the base directory. When compiling the kernel on the memory monitor, the test user programs must be configured before compiling the LegoOS kernel. Next, in order to compile the LegoOS kernel modules, the kernel of both machines being configured must be at the Linux kernel version of 3.11.1. There are numerous websites where this specific kernel can be downloaded and steps to help setup the kernel. The next step is to program the serial output of LegoOS to the virtual machine and not to the host machine. This is necessary because this configuration is using virtual machines. The next step in the configuration is to hard code the LID of the infinibands on each virtual machine used into the LegoOS code. This is necessary because Infiniband requires this information to be provided before accomplishing some of the tasks LegoOS requires. Typically, this information can be acquired when Ethernet is used within the configuration, but LegoOS requires this information to be hard coded, in which, the Infiniband layer can then build off of. The last step of the configuration is to finally install the LegoOS kernel. When installing the kernel, however, the following error occurs which was unable to be resolved in the remainder of this project.



```
[centos@check1 LegoOS]$ sudo make install
scripts/kconfig/conf  --silentoldconfig Kconfig
sh ./arch/x86/boot/install.sh 4.0.0-lego+ arch/x86/boot/bzImage \
System.map "/boot"
depmod: ERROR: could not open directory /lib/modules/4.0.0-lego+: No such file o
r directory
depmod: FATAL: could not search modules: No such file or directory
Kernel version 4.0.0-lego+ has no module directory /lib/modules/4.0.0-lego+
```

Figure 1: Displays error message when installing kernel

From the message, displayed in Figure 1, it can be inferred that it was caused by the Makefile being unable to fully compile the kernel modules. When compiling generic Linux kernel modules, a directory for the modules are created in the /lib/modules directory. However, when compiling the LegoOS kernel modules for this project, no errors occurred in the compilation, but a directory for the LegoOS kernel modules was not created. If one were to replicate this process, it should be noted that this error may occur and should be a starting place for debugging and finding a solution. The LegoOS's kernel attempts to minic a Linux kernel of version 4 or higher. Because of this, an attempt was made to use a basic 4.4.0 Linux kernel modules as a substitute to the LegoOS kernel modules. This solution did not work. The LegoOS repository states to run make in the Linux-modules directory in order to compile the kernel modules. In a typical Linux kernel installation,

```
make modules_install
```

would be ran to compile kernel modules. This script would create the a directory for its kernel modules at the end of the run. Since, LegoOS is based of a typical Linux kernel, an attempt was made to run

```
make modules_install
```

in the LegoOS kernel directory. However, the following error was generated.

```
[centos@check1 LegoOS]$ make
    ↪ modules_install
The present kernel configuration has
    ↪ modules disabled.
Type 'make config' and enable loadable
    ↪ module support.
Then build a kernel with module support
    ↪ enabled.
```

Typically, this error is generated if

```
CONFIG_MODULES
```

is set to no. However, this option is not available within the LegoOS Kconfig file. Whether this is an error that occurred due to a mistake on this process this project took or is a developer error is unknown. This project did make an attempt to reach out to the creators of LegoOS by creating an issue on the LegoOS GitHub repository to resolve this error, but it was not successful. Due to these errors, an attempt was made to find another platform to conduct Coalescent computing.

TABLE 3: Hardware specifications used in GiantVM setup

| **Hardware** | **Vendor** |
|---|---|
| CPU | CPU E5-2670 v3 @ 2.30GHz |
| Server | Intel Xeon |
| Instruction Set | x86-64 |

TABLE 4: Software specifications used in GiantVM setup

| **Software** | **Version** |
|---|---|
| Operating System | Ubuntu16.04 |
| GCC | 5.4.0 |
| Kernel | 4.4.0 |

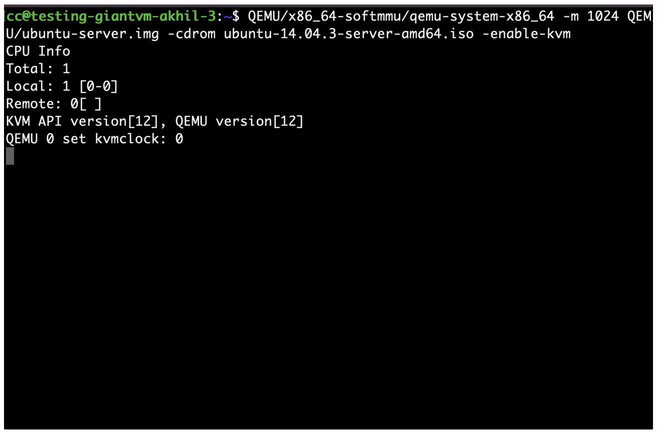
## 3. GiantVM Configuration

GiantVM is a distributed hypervisor that can be built upon any traditional operating system [2]. This allows users to be able to create a distributed system on top of their machines. This project attempted to build GiantVM on top of bare-metal instances rented out from the Chameleon Cloud testbed.

The process into building GiantVM is relatively straightforward. With the hardware and software specifications displayed in tables 3 and 4, a python 2.6 virtual environment is needed in order to compile the distributive QEMU from source. Before going further, the file, interrupt-router.c, must be altered according to the correct network configuration described by GiantVM. This requires adding a path to a common socket for communication on each host machine and adding a TCP port for router communication.

GiantVM was created from altering a stable 2.8 QEMU release. After compiling GiantVM, the distributive Linux kernel, Linux-DSM, must also be compiled and used to boot and install the rest of GiantVM. This is a integral step because Linux-DSM allows for shared memory, the use distributive virtual CPUs, and the timing between systems ([2]). Based upon the system being used to compile and build Linux-DSM, some kernel hacking and changes may need to be made in order to successfully compile Linux-DSM. For this project several configuration changes in the .config file were needed. A hint that an issue related to an incorrect .config script are undefined variables or missing variables related errors. That was a recurring theme when trying to configure Linux-DSM. Once Linux-DSM is installed, reboot the machines being used to configure GiantVM and choose Linux-DSM as your kernel (it will be version 4.9.76+). The next steps in the GiantVM configuration is to create a QEMU image and boot into a virtual-machine the same way as a normal Qemu. For this project, the best way to boot into the GiantVM virtual-machine is to follow the generic instructions for creating and booting into QEMU virtual machine. This conclusion was deduced because, when following the boot instruction given on the GiantVM GitHub repository, the commands lead to an indefinite "hang" of the command. Figure4 depicts this hang.

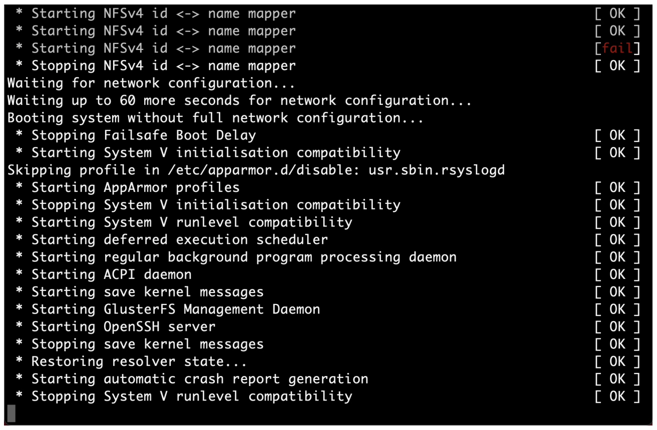
The configuration discussed in this report ends when booting into GiantVM. This is because of a kernel panic that occurs during the boot process of the virtual machine. From this message and from watching the boot process this



```
cc@testing-giantvm-akhil-3:~$ QEMU/x86_64-softmmu/qemu-system-x86_64 -m 1024 QEM
U/ubuntu-server.img -cdrom ubuntu-14.04.3-server-amd64.iso -enable-kvm
CPU Info
Total: 1
Local: 1 [0-0]
Remote: 0[ ]
KVM API version[12], QEMU version[12]
QEMU 0 set kvmclock: 0
```

Figure 2: Visually displaying hanging command

error may be occurring due to failure of a synchronization between the two virtual machines being configured. As can be seen in Figure 3, the boot process began and a kernel was successfully loaded into memory. However, an issue occurs starting a network configuration. In Figure 4, for currently unknown reasons, an emulation message is immediately displayed. From this information, it can be inferred, that the error is Figure 3 is exhibited because the virtual machine from Figure 4 is not at the same step in the boot process as the virtual machine in Figure 3. The GiatVM boot script contains an IP list argument. It was observed that whatever was second in the IP list exhibited the output in Figure 4. Other than being a network configuration error, this error may be caused by an overall communication error. As described before, GiantVM uses shared memory in its configuration as a way to communicate between each virtual machine. This error could be caused by an incorrect configuration of setting up sockets for shared memory communication.



```
 * Starting NFSv4 id <-> name mapper                                      [ OK ]
 * Starting NFSv4 id <-> name mapper                                      [ OK ]
 * Starting NFSv4 id <-> name mapper                                      [fail]
 * Stopping NFSv4 id <-> name mapper                                      [ OK ]
Waiting for network configuration...
Waiting up to 60 more seconds for network configuration...
Booting system without full network configuration...
 * Stopping Failsafe Boot Delay                                           [ OK ]
 * Starting System V initialisation compatibility                         [ OK ]
Skipping profile in /etc/apparmor.d/disable: usr.sbin.rsyslogd
 * Starting AppArmor profiles                                             [ OK ]
 * Stopping System V initialisation compatibility                         [ OK ]
 * Starting System V runlevel compatibility                               [ OK ]
 * Starting deferred execution scheduler                                  [ OK ]
 * Starting regular background program processing daemon                  [ OK ]
 * Starting ACPI daemon                                                   [ OK ]
 * Starting save kernel messages                                          [ OK ]
 * Starting GlusterFS Management Daemon                                   [ OK ]
 * Starting OpenSSH server                                                [ OK ]
 * Stopping save kernel messages                                          [ OK ]
 * Restoring resolver state...                                            [ OK ]
 * Starting automatic crash report generation                             [ OK ]
 * Stopping System V runlevel compatibility                               [ OK ]
```

Figure 3: Output of one virtual machine

## 4. Conclusion

This report discussed the process to configuring GiantVM and LegoOS using resources of provided by the Chameleon Cloud testbed. Though both configurations were unsuccessful, important errors and potential bugs were discussed and solutions to tricky ones were confirmed. This guide will be useful to anybody working on these ideas to help understand the issues. It is the goal of this paper to eventually lead to the solution of the final issues in the configuration of both of the distributed platforms discussed. Both of the platforms are important due to their potential use of a Coalescent Computing platform and the hope is that they can be used to create a platform for Coalescent Computing related tests.

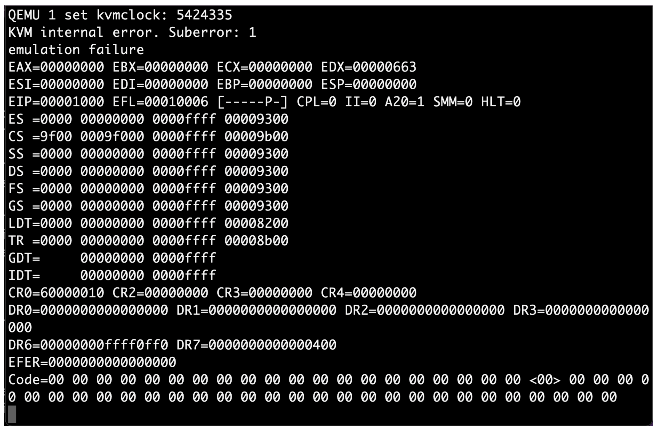
## References

[1] Kyle C. Hale. "Coalescent Computing". In: (2021). arXiv: 2104.07122 [cs.DC].

[2] Zhang Jin et al. "GiantVM: a type-II hypervisor implementing many-to-one virtualization". In: Proceedings of the 16th ACM SIGPLAN/SIGOPS International Co Association for Computing Machinery. 2020, pp. 30–44.

[3] Joe Mambretti, Jim Chen, and Fei Yeh. "Next generation clouds, the chameleon cloud testbed, and software defined networking (sdn)". In: 2015 International Conference on Cloud Computing Research and IEEE. 2015, pp. 73–79.

[4] Shan Yizhou et al. "LegoOS: A disseminated, distributed OS for hardware resource disaggregation". In: Proceedings of the 13th USENIX Symposium on Operating Systen USENIX. 2018, pp. 69–87.



```
QEMU 1 set kvmclock: 5424335
KVM internal error. Suberror: 1
emulation failure
EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000663
ESI=00000000 EDI=00000000 EBP=00000000 ESP=00000000
EIP=00001000 EFL=00010006 [-----P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0000 00000000 0000ffff 00009300
CS =9f00 0009f000 0000ffff 00009b00
SS =0000 00000000 0000ffff 00009300
DS =0000 00000000 0000ffff 00009300
FS =0000 00000000 0000ffff 00009300
GS =0000 00000000 0000ffff 00009300
LDT=0000 00000000 0000ffff 00008200
TR =0000 00000000 0000ffff 00008b00
GDT=     00000000 0000ffff
IDT=     00000000 0000ffff
CR0=60000010 CR2=00000000 CR3=00000000 CR4=00000000
DR0=0000000000000000 DR1=0000000000000000 DR2=0000000000000000 DR3=0000000000000
000
DR6=00000000ffff0ff0 DR7=0000000000000400
EFER=0000000000000000
Code=00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 <00> 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 4: Output of kernel panic of other virtual machine