SIT315 M3.S1P: Distributed Computing

Codey Funston | S222250824
No Group

Activity 1 – Distributed Computing

1. Scaling-out a system is increasing the total resources which can be compute or memory, by distributing the load across multiple computers. Such as distributing a database across multiple servers. Scaling-up is for the same purpose but instead the original machine has its hardware upgraded such as a more powerful CPU or extra RAM or storage. An example of this could be in video gaming where you don't want to upgrade the entire computer, so you replace the graphic card with a better one. It is often easier and cheaper to scale out, especially with automation software or cloud computing, and there is also no down-time required for hardware upgrades like with scaling up.

2. Distributed databases are like standard databases to APIs or admins, but they have their data spread across multiple locations. The locations are connected usually by high-speed networking setups in data centres to give the appearance of standard singular database. I think a cool way to think about it, as well as the other following types of distributed systems, is like memory pages. When a user is interacting with an operating system in the user space such as when writing a program with memory management, they see memory as contiguous blocks of data when behind the scenes the kernel is actually allocating memory in a way that would be super confusing if we had to deal with it. Distributed computing is very similar to parallel programming on a single CPU, however, each thread is instead a single computer and instead of communicating between the threads within one computer, networking is used to send messages. Distributed file systems are like distributed databases, except that it is data stored in file system format instead of rows and columns or documents like for databases. Distributed file systems also appear regular to anyone interacting with them. Distributed applications are a very popular way of designing modern systems whereby certain concerns of the overall application are split into modules called microservices which can be individually scaled based on demand for certain parts of the system.

3.

    1. Security: A big challenge in distributed systems is maintaining security now that communication isn't just done within a single computer. By using network communication, it allows for malicious actors to intercept data, pose as members in the system, and steal data. To help prevent this there are measures such as encryption and identity authentication.

    2. Concurrency/Synchronisation: Just like with parallel programming with multiple threads and shared data, each part of a distributed system may need to be synchronised for various reasons. Some important scenarios are transactional systems like electronic banking, as well as databases that follow the ACID protocol.

3. Transparency/abstraction: For a distributed system to be designed well and work smoothly you want to split it into layers of abstraction just like with functions and classes in programming. For example, you wouldn't want an average user of OneDrive or Dropbox to have to allocate files to certain servers when they can see one getting close to full.

4. Heterogeneity: Since logic is executed or data is stored on different hardware and operating systems there needs to be ways to guarantee that it will run or be stored in the correct format. For example, even though a distributed system would be unlikely to run across Windows and Mac machines their file storage difference is a good way to illustrate the issue. Windows uses the NTFS file system mostly, whereas Mac uses APFS. So, when you transfer files between them there needs to be a conversion process at some point along the way. In real distributed systems the issue of heterogeneity/compatibility is often solved by containerising applications with software such as Docker.

5. Fault Tolerance: With a distributed system you don't want it to fail overall just because one aspect/node did. This is what network routing was invented for to allow multiple paths to be taken between network endpoints.

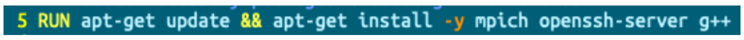
6. Scalability: Regardless of what type of scaling method is used there should be automation in place to ensure a smooth transition as the system's capabilities upgrade. For example, in AWS you can set auto-scaling to occur based on policies such as when the average system load passes a certain value.

7. Openness: This concept has been proven to be beneficial in systems with external communication and heterogeneity such as computer networks. If there wasn't a lot of openness with protocols and network card hardware it would make it very hard to communicate between devices of different manufactures. A big challenge with this in distributed systems is being able to secure standards because when they are open, hackers can take advantage of them.

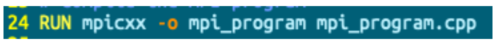
Activity 2 – Hello MPI

Since my machine is an Apple Silicon based Mac I used Docker containers to as nodes. I followed steps very similar to those in the provided steps to run MPI distributed solution with Docker word document.

1. First, I created a directory with the Dockerfile, hosts text file, and the provided MPI C++ program. The Dockerfile allowed for password-less SSH communication between containers and installed the required software which was mpich and Open SSH. Since the provided Dockerfile was intended for C I had to add g++ to the install list, and compile with mpicxx instead of mpicc.

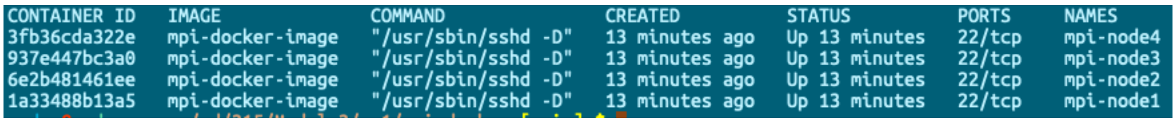


```
RUN apt-get update && apt-get install -y mpich openssh-server g++
```



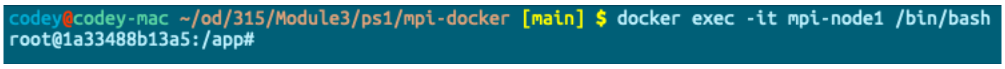
```
RUN mpicxx -o mpi_program mpi_program.cpp
```

2. Next, I built the image in the current directory and ran 4 containerised instances of it for the nodes in the MPI program.



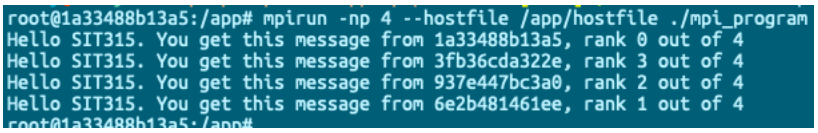
```
CONTAINER ID   IMAGE              COMMAND               CREATED          STATUS          PORTS     NAMES
3fb36cda322e   mpi-docker-image   "/usr/sbin/sshd -D"   13 minutes ago   Up 13 minutes   22/tcp    mpi-node4
937e447bc3a0   mpi-docker-image   "/usr/sbin/sshd -D"   13 minutes ago   Up 13 minutes   22/tcp    mpi-node3
6e2b481461ee   mpi-docker-image   "/usr/sbin/sshd -D"   13 minutes ago   Up 13 minutes   22/tcp    mpi-node2
1a33488b13a5   mpi-docker-image   "/usr/sbin/sshd -D"   13 minutes ago   Up 13 minutes   22/tcp    mpi-node1
```

3. By accessing the network settings of the containers I returned each node's IP address, they were the default types of 172.17.0.{2..5}. I added these to the hosts file of the master node which was node1 in this case, then entered into its shell.



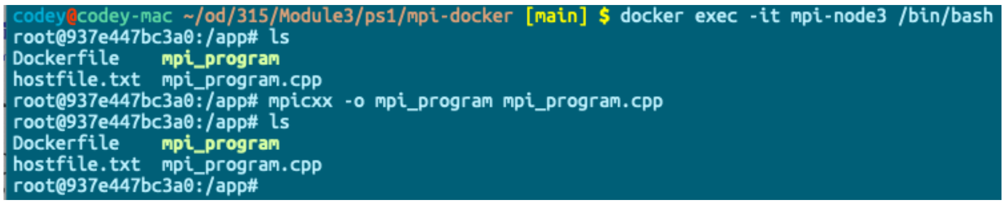
```
codey@codey-mac ~/od/315/Module3/ps1/mpi-docker [main] $ docker exec -it mpi-node1 /bin/bash
root@1a33488b13a5:/app#
```

4. Once in to the master node I ran the MPI program which started the master node itself and all the slave nodes. This ran successfully with 4 unique ranks printed to the terminal.



```
root@1a33488b13a5:/app# mpirun -np 4 --hostfile /app/hostfile ./mpi_program
Hello SIT315. You get this message from 1a33488b13a5, rank 0 out of 4
Hello SIT315. You get this message from 3fb36cda322e, rank 3 out of 4
Hello SIT315. You get this message from 937e447bc3a0, rank 2 out of 4
Hello SIT315. You get this message from 6e2b481461ee, rank 1 out of 4
```

5. Alternatively, I could have opened the shell of each container and gone in and compiled the programs manually like so:



```
codey@codey-mac ~/od/315/Module3/ps1/mpi-docker [main] $ docker exec -it mpi-node3 /bin/bash
root@937e447bc3a0:/app# ls
Dockerfile    mpi_program
hostfile.txt  mpi_program.cpp
root@937e447bc3a0:/app# mpicxx -o mpi_program mpi_program.cpp
root@937e447bc3a0:/app# ls
Dockerfile    mpi_program
hostfile.txt  mpi_program.cpp
root@937e447bc3a0:/app#
```