Getting started with the Confidential Consortium Framework on Azure

A mini guide for developers

Version 0.1, April 2020

For the latest information about Azure, please see
https://azure.microsoft.com/en-us/overview/

For the latest information on Azure Confidential Computing (ACC), please see
https://azure.microsoft.com/en-us/solutions/confidential-compute/

For the latest information about open source on Azure, please see
https://azure.microsoft.com/en-us/overview/choose-azure-opensource/

For the latest information on the Open Enclave (OE) SDK, please see
https://openenclave.io/sdk/

For the latest information on the Confidential Consortium Framework (CCF), please see
https://aka.ms/ccf/

This page is intentionally left blank.

Table of contents

Notice

This mini guide for developers is intended to new way for companies to build and execute blockchain-type or any other distributed applications using the Confidential Consortium Framework (CCF). The Confidential Consortium Framework (CCF) is available in open source at https://microsoft.github.io/CCF/.

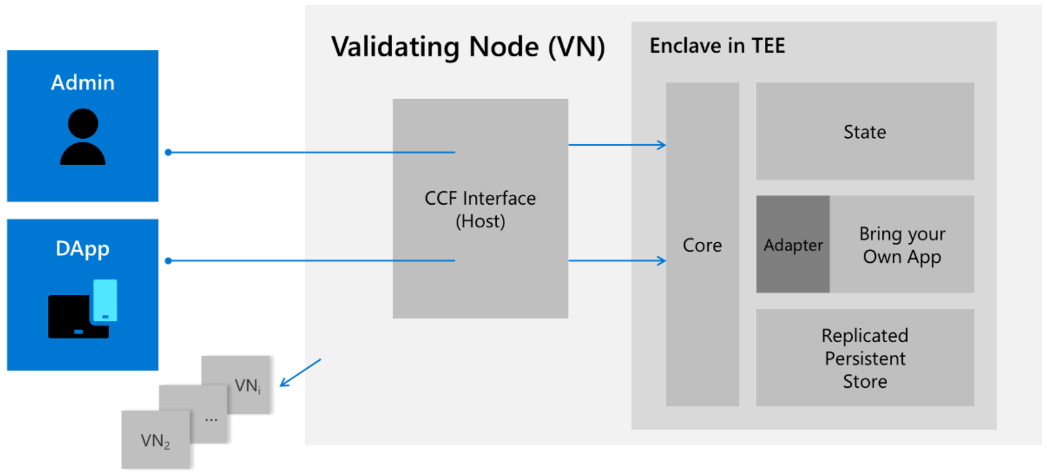
Publication: April 2020
Version 0.1

About this guide

Welcome to the **Getting started with the Confidential Consortium Framework on Azure** mini guide for developers.

This document is part of a series of guides that covers confidential computing in the Cloud, and the Edge, and considerations that pertain to it from a development perspective and/or an infrastructure one. This series of guides is available at https://aka.ms/CCDevGuides.

The Confidential Consortium Framework[1] (CCF), a joint project between Microsoft Research and Azure Engineering, is an open-source framework for building a new category of secure, highly available, and performant distributed applications that focus on multi-party compute and data. While not limited just to blockchain applications, CCF can enable high-scale, confidential blockchain networks that meet key enterprise requirements, providing a means to accelerate production enterprise.



Note For an overview, watch the episodes Confidential Consortium Framework (CCF) Overview[2] and Confidential Consortium Framework (CCF) Part II: A deeper look and demos[3] on the Block Talk series on Channel 9.

Deployed as a blockchain-type network between parties, every node of a CCF network is running a Trusted Execution Environment[4] (TEE) or enclave, - you can read the guide **Building and Executing Trusted Execution Environment (TEE) based applications on Azure** in this series of guides to learn about TEE -.

CCF is built on top of the Microsoft Open Enclave SDK[5] (OESDK), (yet) another open source framework available on GitHub over two years. The OESDK aims at creating a single unified enclaving abstraction and consistent API

[1] Confidential Consortium Framework: https://aka.ms/ccf

[2] Confidential Consortium Framework (CCF) Overview: https://channel9.msdn.com/Shows/Blocktalk/Confidential-Consortium-Framework-CCF-Overview

[3] Confidential Consortium Framework (CCF) Part II: A deeper look and demos: https://channel9.msdn.com/Shows/Blocktalk/Confidential-Consortium-Framework-CCF-Part-II-A-deeper-look-and-demos

[4] Trusted execution environment: https://en.wikipedia.org/wiki/Trusted_execution_environment

[5] Open Enclave SDK: https://Open Enclave.io/sdk/

surface for developers to build applications once that run across multiple TEE architectures, technologies and platforms[6], such as the Intel Software Extension Guard[7] (SGX).

CCF also provides a simple programming model of a highly available data store and a universally verifiable log that implements a ledger abstraction. Furthermore, CCF leverages trust in a consortium of governing members and in a network of replicated hardware-protected execution environments to achieve high throughput, low latency, strong integrity and strong confidentiality for application data and code executing on the ledger.

In this mini guide, and as its title suggest, we will cover the basics of the CCF.

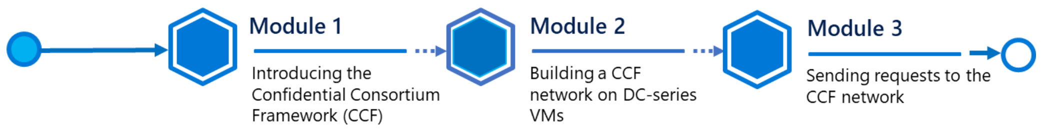
This guide aims at helping you as a developer start with CCF, and in particular build a trusted network of nodes/TEEs, deploy an application on top of it, etc.

You will indeed learn how to create and deploy with CCF this new kind of multi-party applications on top of Azure Confidential Computing[8] (ACC), and the SGX-based DC-series family of VMs it provides.

For that purposes, you're invited to follow a short series of modules, each of them illustrating a specific aspect of the TEE-based application development.

Please note, that besides Microsoft Azure here, networks can be run on-premises, in one or many cloud-hosted data centers, or in any hybrid configuration.

Each module within the guide builds on the previous. You're free to stop at any module you want, but our advice is to go through all the modules.


Module 1
Introducing the Confidential Consortium Framework (CCF)
Module 2
Building a CCF network on DC-series VMs
Module 3
Sending requests to the CCF network

At the end of the mini guide, you will be able to:

- Instantiate DC-series VMs well-suited for blockchain or any other trusted distributed multi-party applications development,
- Install the Confidential Consortium Framework (CCF) on DC-series VMs,
- Deploy and manage a multi-node CCF network.

Guide elements

In the mini guide modules, you will see the following elements:

- **Step-by-step directions**. Click-through instructions - along with relevant snapshots - or links to online documentation for completing each procedure or part.
- **Important concepts**. An explanation of some of the concepts important to the procedures in the module, and what happens behind the scenes.

[6] The future of computing: intelligent cloud and intelligent edge: https://azure.microsoft.com/en-us/overview/future-of-cloud

[7] Intel Software Extension Guard: https://software.intel.com/en-us/sgx

[8] Azure Confidential Computing: https://azure.microsoft.com/solutions/confidential-compute/

- **Sample applications, and files**. A downloadable or cloneable version of the project containing the code that you will use in this guide, and other files you will need. **Please go to https://github.com/Microsoft/CCF on GitHub to download or clone all necessary assets.**

Guide prerequisites

To successfully leverage the provided code in this starter guide, you will need:

- A Microsoft account[9].
- An Azure subscription. If you don't have an Azure subscription, create a free account[10] before you begin.
- A windows 10 local machine.
- A code editor of your choice, such as Visual Studio[11] or Visual Studio Code[12], with C++ for Linux and Open Enclave installed. The related installation and configuration will be further covered later in this guide.
- A terminal console for your Windows 10 local machine, which allows you to remotely connect to a virtual machine (VM) in SSH, such as PuTTY[13], Git for Windows[14] (2.10 or later).

Important note With Git, ensure that long paths are enabled: `git config --global core.longpaths true`.

As far as the latter is concerned, recent versions of Windows 10 provide OpenSSH client commands to create and manage SSH keys and make SSH connections from a command prompt.

Note For more information, see blogpost WHAT'S NEW FOR THE COMMAND LINE IN WINDOWS 10 VERSION 1803[15].

Installing OpenSSH on Windows 10

The OpenSSH Client and OpenSSH Server are separately installable components in Windows 10 1809 and above.

Note For information about the OpenSSH availability on Windows 10, see here.

To install OpenSSH on your Windows 10 local machine, perform the following steps.

1. Open an elevated PowerShell console.
2. Run the following command:

```
PS C:\> Add-WindowsCapability -Online -Name OpenSSH.Client~~~~0.0.1.0
```

[9] Microsoft Account: https://account.microsoft.com/account?lang=en-us

[10] Create your Azure free account today: https://azure.microsoft.com/en-us/free/?WT.mc_id=A261C142F

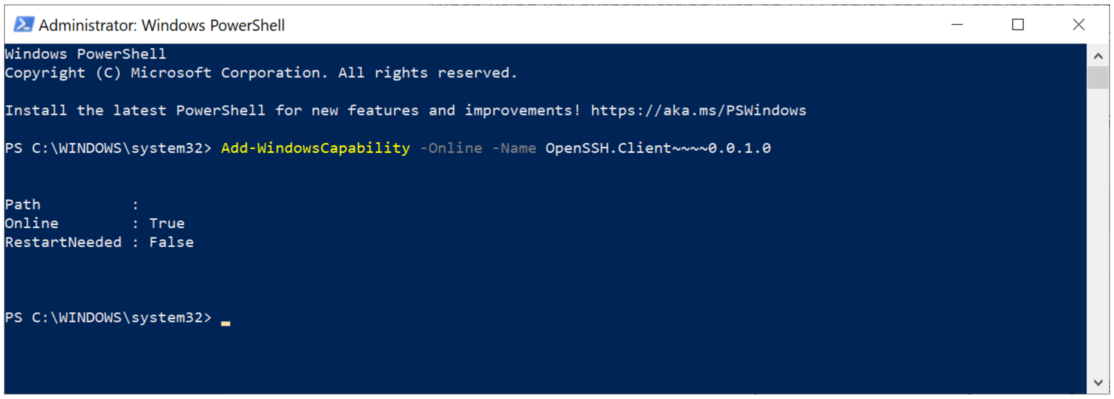
[11] Visual Studio: https://visualstudio.microsoft.com/

[12] Visual Studio Code: https://code.visualstudio.com/

[13] PuTTY: https://www.chiark.greenend.org.uk/~sgtatham/putty/

[14] Git for Windows: https://git-for-windows.github.io/

[15] WHAT'S NEW FOR THE COMMAND LINE IN WINDOWS 10 VERSION 1803: https://blogs.msdn.microsoft.com/commandline/2018/03/07/windows10v1803/



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\WINDOWS\system32> Add-WindowsCapability -Online -Name OpenSSH.Client~~~~0.0.1.0


Path          :
Online        : True
RestartNeeded : False



PS C:\WINDOWS\system32> _
```

Once the installation completes, you can use the OpenSSH client from PowerShell or the Windows 10 command shell.

Generating your RSA Key pairs with OpenSSH

OpenSSH includes different tools and more specifically the `ssh-keygen` command for generating secure RSA key pairs, that can be in turn used for key authentication with SSH.

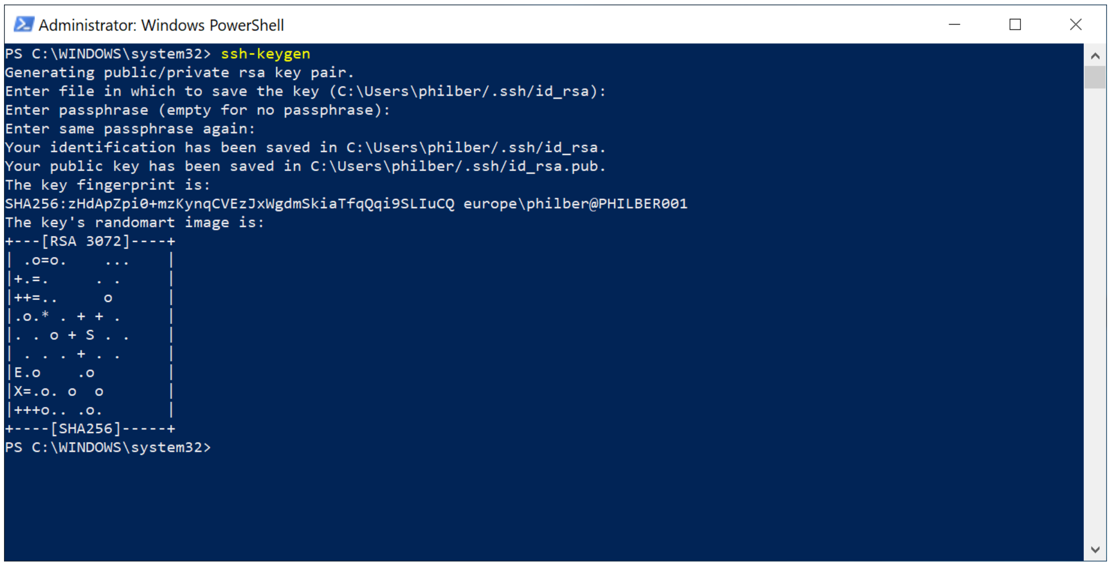
RSA Key pairs refer to the public and private key files that are used by certain authentication protocols.

To generate your RSA Key pairs, perform the following steps.

1. Open an elevated PowerShell console.
2. Run the following command:

```
PS C:\> ssh-keygen
```

You can just hit ENTER to generate them, but you can also specify your own filename if you want. At this point, you'll be prompted to use a passphrase to encrypt your private key files. The passphrase works with the key file to provide 2-factor authentication. For this example, we are leaving the passphrase empty.



```
PS C:\WINDOWS\system32> ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (C:\Users\philber/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in C:\Users\philber/.ssh/id_rsa.
Your public key has been saved in C:\Users\philber/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:zHdApZpi0+mzKynqCVEzJxWgdmSkiaTfqQqi9SLIuCQ europe\philber@PHILBER001
The key's randomart image is:
+---[RSA 3072]----+
| .o=o.    ...    |
|+.=.     . .     |
|++=..     o      |
|.o.* . + + .     |
|. . o + S . .    |
| . . . + . .     |
|E.o    .o        |
|X=.o. o  o       |
|+++o.. .o.       |
+----[SHA256]-----+
PS C:\WINDOWS\system32>
```

Note SSH public-key authentication uses asymmetric cryptographic algorithms to generate two key files – one "private" and the other "public". The **private key** file is the equivalent of a password and should protected under all circumstances. If someone acquires your private key, they can log in as you to any SSH server you have access to. The **public key** is what is placed on the SSH server and may be shared without compromising the private key.

When using key authentication with an SSH server, the SSH server and client compare the public key for username provided against the private key. If the public key cannot be validated against the client-side private key, authentication fails.

By default, the files are saved in the following folder *%USERPROFILE%\.ssh:*

File	Description
%USERPROFILE%\.ssh\id_rsa	Contains the RSA private key
%USERPROFILE%\.ssh\id_rsa.pub	Contains the RSA public key.

Module 1: Introducing the Confidential Consortium Framework (CCF)

Before starting to create your Confidential Consortium Framework (CCF) network nodes, let's take the time to (shortly) remind or present you what blockchain (network) is and how TEEs are keeping the network safe from fraudulent transactions or external attacks in CCF.

This is the purpose of this first module to introduce some important concepts regarding blockchain network, and in contrast/parallel, outlines the benefits CCF provide. If you already familiar with these considerations, you can skip this module and jump into the next one start building your own network.

A brief recap on blockchain network

A blockchain (network) is a tamper-proof, highly available, decentralized ledger.

It's a system that maintains cryptographically chained blocks of transactions (containing cryptocurrencies, code or assets) across nodes that are linked in a peer-to-peer network:

- It records **what happened** and the **order** it happened in.
- There isn't any central server or organization, which rules the network. It's a fully decentralized environment.

There are 2 types of blockchain: **public** vs. **private**.

On a public blockchain, members are usually unknown - you only know public addresses - and everybody can join the network. This is a **permissionless network**.

Conversely, on a private blockchain, everybody knows each other and when someone tries to join the blockchain, he usually needs to be accepted inside the network by other peers. This is a **permissioned network**.

To keep the network tamper-proof, a blockchain is usually based on/govern by algorithms. The goal of those algorithms aims at validating that a transaction is valid and spreading this transaction across the peer network. Indeed, every peer of the network runs its own copy of the transaction ledger, so such an algorithm is in charge of keeping every ledger consistent across nodes.

On a public blockchain, the algorithm requires that a peer, which wants to validate a transaction, needs to bet a lot of "something" (computation power, cryptocurrency, etc.). So, considering that, at least half of the members inside the network are not trying to attack or steal something on the network, this part of peers will overwhelm cheaters, which will lost their "bets". This is why a public blockchain can be considered as secure: if you try to attack the blockchain, you likely will lose more than what you could win as potential rewards.

Those algorithms such as the well-known Proof of work[16] (PoW) or Proof of stake[17] (PoS), provide reliability but there are some drawbacks, such as a low transaction throughput, a high computation cost, or an impossibility to scale up. To illustrate the point, Bitcoin currently uses approximately 69 TWh/year of electricity, about the same as

[16] Proof of work: https://en.wikipedia.org/wiki/Proof_of_work

[17] Proof of stake: https://en.wikipedia.org/wiki/Proof_of_stake

Austria, to process between 2 and 3 transactions (Tx)/second. Determining if a transaction was successful is probabilistic rather than deterministic and takes at least an hour. In addition, transactions are not confidential.

This is (currently) not "enterprise ready", resulting in the rise of private or consortium blockchains (in the meantime). In so far as you know every member inside the network, you can implement lighter algorithms (with less confidence) to verify transactions. Indeed, if someone tries to cheat, he/she will simply be kicked out of the blockchain. But this is not a perfect solution either, as this is still possible to cheat at least one time before being kicked, or, even if performances of the network are better, this isn't as good as a database.

Furthermore, governance is a difficult problem, making changes and bug fixes problematic.

A consortium first approach with CCF

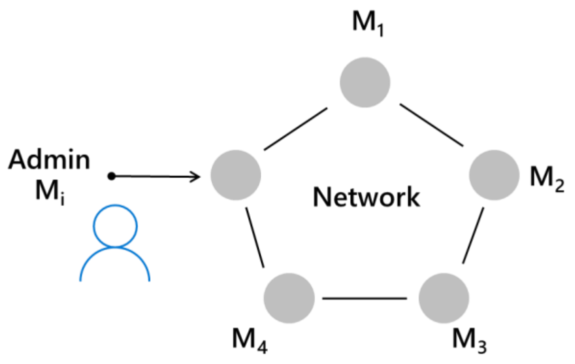
In this context, Microsoft tried another approach: what if the algorithm were just a way to establish a consensus between nodes, and the security part would be managed by something else, such as Trusted Execution Environments (TEEs) or enclaves.

As you saw before, an enclave is a secure area inside the processor, in a fully isolated environment. A user connected to the machine (even an administrator) can't see what is running and processing inside this enclave. However, the user can get an attestation about the code running in that enclave. This means that if you put some (trusted) code inside an enclave, you can verify at any moment if your code was modified or not.

And this is basically how CCF works: by putting the code in charge of verifying transactions inside enclaves, you can be sure that a transaction signed by the code inside the enclave and the code itself are correct and not fraudulent.

Also, since you can verify transaction by yourselves, you can make them private if needed, as other nodes, which will append their ledger with this transaction, will only have to know if the transaction is signed, they don't have to know about their content anymore.

With the use of Trusted Execution Environments (TEEs) or enclaves, CCF creates a trusted distributed blockchain network of physical nodes on which to run a distributed ledger, providing secure, reliable components for the protocol to use.



CCF is an infrastructure framework that leverages HW based (e.g. Intel SGX via ACC) Trusted Execution Environments (TEEs), Paxos[18] and similar consensus mechanisms, a governance model, standard cryptography to

[18] Paxos (computer science): https://en.wikipedia.org/wiki/Paxos_%28computer_science%29

enable arbitrary blockchain protocols to deliver high-throughput, secure computation with confidential transaction models for consortium blockchain networks.

The Confidential Consortium Framework (CCF) doesn't compete with blockchain's technologies. It rather provides the foundation, the groundwork for turning these technologies into high-performance, confidential implementations.

All of these simplify consensus and transaction processing for high throughput and confidentiality.

Leveraging the power of trusted execution environments (TEEs), decentralized systems concepts, and cryptography, CCF provides for blockchain and multi-party application builders with enterprise-ready computation or blockchain networks that deliver:

- **Throughput and latency approaching database speeds.** Through its use of TEEs, CCF creates a network of remotely attestable enclaves - you can read the guide **Leveraging Attestations with Trusted Execution Environment (TEE) based applications on Azure** in this series of guides to learn about TEE -.

  This gives a Web of Trust across the distributed system, allowing a user that verifies a single cryptographic quote from a CCF node to effectively verify the entire network. This simplifies consensus and thus improves transaction speed - currently running 50,000+ Tx/sec. For context, Visa averages 2,000 Tx/sec, with 50,000 Tx/sec peaks - and latency - Transaction latency in 10s of milliseconds -, all without compromising security or assuming trust.
- **Richer, more flexible confidentiality models.** Beyond safeguarding data access with encryption-in-use via TEEs, CCF uses industry standards (mTLS and remote attestation) to ensure secure node communication. Transactions can be processed in the clear or revealed only to authorized parties, without requiring complicated confidentiality schemes. Confidential transactions is more than welcome in a number of multi-party scenarios.
- **Network and service policy management through non-centralized governance.** CCF provides a network and service configuration to express and manage consortium and multi-party policies. Strong, smart contract-based governance actions, such as adding members to the governing consortium or initiating catastrophic recovery, can be managed and recorded through standard ledger transactions agreed upon via stakeholder voting.
- **Improved efficiency versus traditional blockchain networks.** CCF improves on bottlenecks and energy consumption by eliminating computationally intensive consensus algorithms for data integrity, such as proof-of-work or proof-of-stake.
- **Attestation at scale**. CCF Maintains integrity, resilience, and accountability properties of existing blockchains' technologies.

A consortium first approach in CCF

In a public blockchain network, anyone can transact on the network, actors on the network are pseudo-anonymous and untrusted, and anyone can add nodes to the network — with full access to the ledger and with the ability to participate in consensus. Similarly, other distributed data technologies (such as distributed databases) can have challenges in multi-party scenarios when it comes to deciding what party operates it and whether that party could choose or could be compelled to act maliciously.

In contrast, in a consortium or multi-party network backed by TEEs, as enabled by CCF, consortium member identities and node identities are known and controlled. A trusted network of TEEs, or enclaves, running on physical nodes is established without requiring the actors that control those nodes to trust one another, i.e. what code is run is controlled and correctness of its output can be guaranteed, simplifying the consensus methods and reducing duplicative validation of data.

In CCF, using TEE technology through all the goodness of the OESDK (in terms of abstraction level and unified API surface), the enclave of each node in the network (where cryptographically protected data is processed) can decide whether it can trust the enclaves of other nodes based on mutual attestation exchange and mutual authentication, regardless of whether the parties involved trust each other or not. This enables a network of verifiable, remotely attestable enclaves on which to run a distributed ledger and execute confidential and secure transactions in highly performant and highly available fashion.

If you want to learn more about CCF, we strongly advise to read the CCF technical report CCF: A Framework for Building Confidential Verifiable Replicated Services[19].

Also feel free to explore the GitHub repository available here[20], or the website at this address.[21]

Now that you're equipped with an understanding of the core capabilities of CCF, you are ready to build our own network, based on that great framework.

This concludes this brief introduction.

[19] CCF: A Framework for Building Confidential Verifiable Replicated Services: https://github.com/microsoft/CCF/blob/master/CCF-TECHNICAL-REPORT.pdf

[20] Confidential Consortium Framework: https://github.com/microsoft/CCF

[21] Welcome to CCF's documentation: https://microsoft.github.io/CCF/

Module 2: Building a CCF network on DC-series VMs

You will see in this second module how to create CCF nodes based on the DC-series VMs available in Azure Confidential Computing (ACC).

This section covers the following two activities:

1. Deploying a 2-node network.
2. Installing CCF on your nodes.

Each activity is described in order in the next sections.

Deploying a 2-node network

For this sake of this guide, you will create a simple 2-node blockchain network. As CCF runs on top of the Open Enclave SDK (OESDK), a node will unsurprisingly be a DC-series VM which is an easy to instantiate enclave-capable VM in Azure.

The DC-series corresponds to the initially introduced family of Confidential Computing (CC) VMs in Azure, that are backed by the 3.7GHz Intel XEON E-2176G[22] processor with SGX technology. This family is currently available in East US and West Europe regions only.

For the sake of this guide, and to minimize the implied cost, you can opt for the Standard_DC2s with 2 vCPUs and 8 GB of memory, between the two available size.

Furthermore, in terms of OS, amongst the three operating systems are supported for the above family of VMs, you will choose Ubuntu Server 18.04 TLS.

Please also ensure NOT to include the Open Enclave SDK (OESDK) with your DC_series VMs deployment. **No** should be selected. As of this writing, the version of the OESDK that comes with the DC_series VM is currently outdated. You will have to rather install it manually later on.

> **Note** For more information on how to install the Open Enclave SDK, see article INSTALL THE OPEN ENCLAVE SDK (UBUNTU 18.04)[23].

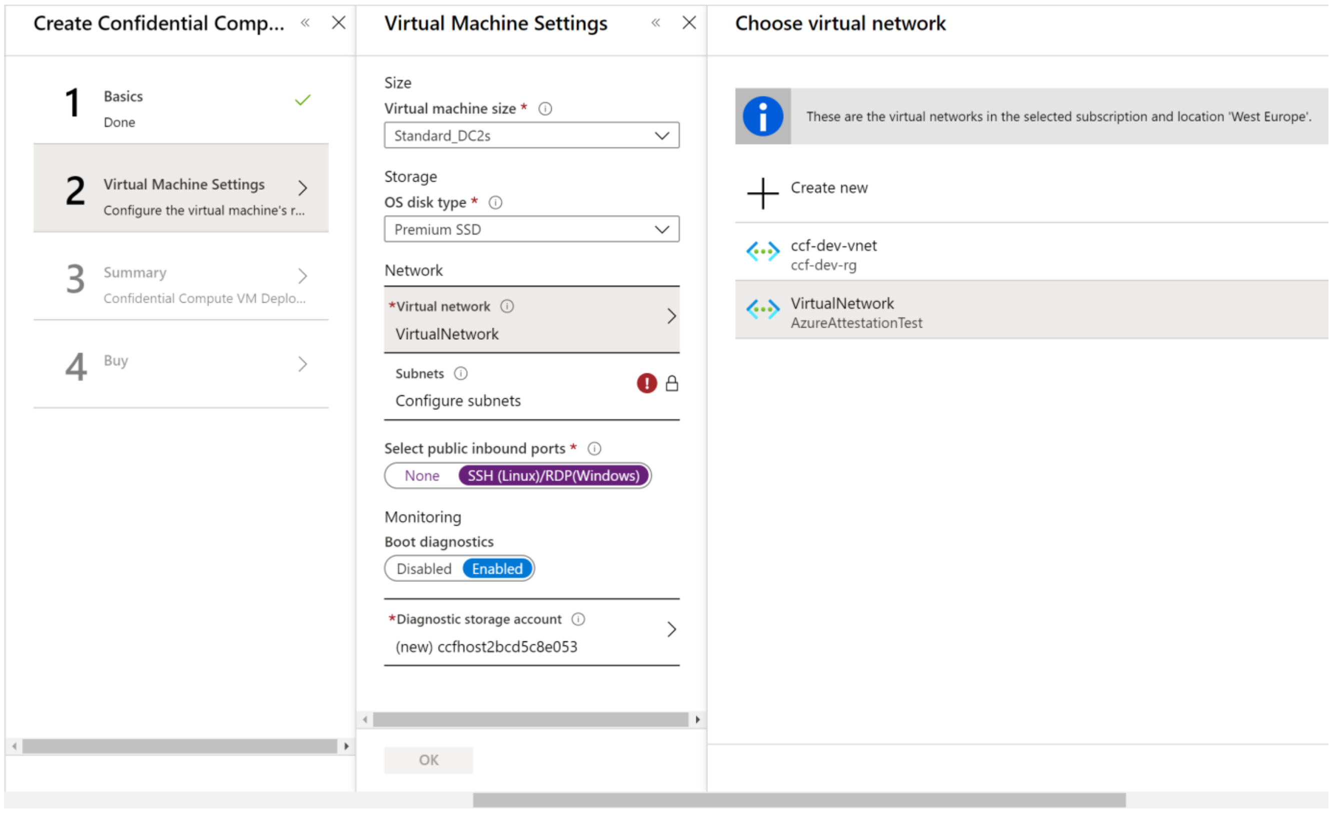
As such, the 2-node network requires 2 instances of the DC-series.

[22] Intel XEON E-2176G: https://www.intel.com/content/www/us/en/products/processors/xeon/e-processors/e-2176g.html

[23] INSTALL THE OPEN ENCLAVE SDK (UBUNTU 18.04): https://github.com/openenclave/openenclave/blob/v0.6.x/docs/GettingStartedDocs/install_oe_sdk-Ubuntu_18.04.md

You can refer to the section § *Module1: Setting up a confidential computing VM in Azure* of the **guide Building and Executing Trusted Execution Environment (TEE) based application in Azure** in this series of guide to see how to instantiate such VMs.

Although you will need to create a new resource group or point to an empty one, make sure that you select the same virtual network (VNet) for your two VMs when instantiating it, e.g. **ccf-dev-vnet** in our illustration.


Create Confidential Comp...
1 Basics
Done
2 Virtual Machine Settings
Configure the virtual machine's r...
3 Summary
Confidential Compute VM Deplo...
4 Buy
Virtual Machine Settings
OK
Choose virtual network
Create new

Size
Virtual machine size * Standard_DC2s
Storage
OS disk type * Premium SSD
Network
*Virtual network VirtualNetwork
Subnets Configure subnets
Select public inbound ports * None [✓ SSH (Linux)/RDP(Windows)]
Monitoring
Boot diagnostics Disabled [✓ Enabled]
*Diagnostic storage account (new) ccfhost2bcd5c8e053

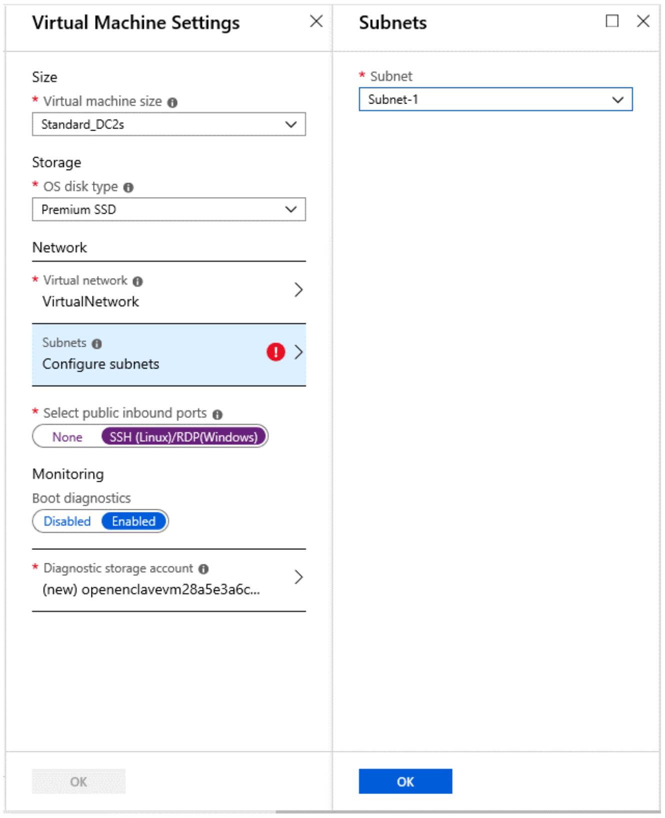
These are the virtual networks in the selected subscription and location 'West Europe'.

ccf-dev-vnet
ccf-dev-rg

VirtualNetwork
AzureAttestationTest

You can then place the VM on the same subnet:

1. Click on **Configure subnets**.
2. Select the subnet, i.e. **Subnet-1** in our illustration and click on **OK**.


Virtual Machine Settings

Size
* Virtual machine size: Standard_DC2s

Storage
* OS disk type: Premium SSD

Network
* Virtual network: VirtualNetwork

Subnets: Configure subnets

* Select public inbound ports: None [✓ SSH (Linux)/RDP(Windows)]

Monitoring
Boot diagnostics: Disabled [✓ Enabled]

* Diagnostic storage account: (new) openenclavevm28a5e3a6c...

Subnets
* Subnet: Subnet-1

Proceed with the rest of the configuration.

Once completed, you are now ready to install CCF on your two nodes.

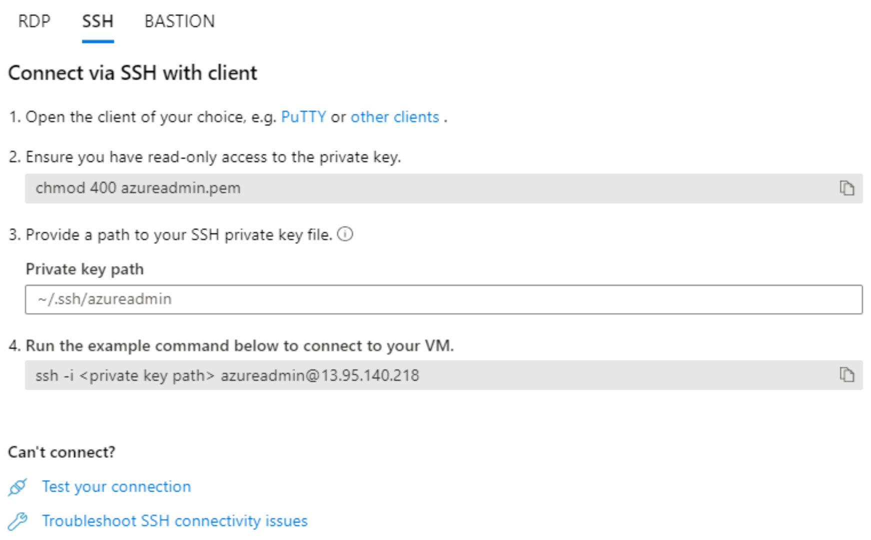
Connecting to your nodes

Once your VMs are online, by using a SSH client of your choice, such as OpenSSH, PuTTY, etc. you can test your remote connection to the newly created VM using the administrator credentials provided above. The public IP address of the VM can be found on the VM Networking page.

Note Depending on your configuration, you may need to configure a proxy in the SSH client to connect to the virtual machine.

To connect to your one of your VMs using OpenSSH on your Windows 10 local machine, perform the following steps:

1. From the Azure portal, search for your VM and click on it to display its menu.
2. Click on **Connect**, select SSH, and then make a note of the public IP address and the SSH connection string.



Connect via SSH with client

1. Open the client of your choice, e.g. PuTTY or other clients .
2. Ensure you have read-only access to the private key.

```
chmod 400 azureadmin.pem
```

3. Provide a path to your SSH private key file.

Private key path

~/.ssh/azureadmin

4. Run the example command below to connect to your VM.

```
ssh -i <private key path> azureadmin@13.95.140.218
```

Can't connect?

Test your connection

Troubleshoot SSH connectivity issues

In our illustration, the IP address of the DC_series VM is 13.95.140.218, and the SSH connection string is azureadmin@13.95.140.218.

3. Open a PowerShell console, and the SSH to your VM:

```
PS C:\> ssh azureadmin@13.95.140.218
```

4. When prompted, type "*yes*". Optionally specify your passphrase if any for your private key.

Et voila! You should now be connected to one of your DC-series VMs.

Installing CCF on your nodes

In your Azure subscription, you should now have two DC-series VMs and a common virtual network that connect them together, e.g. in our illustration: **ccf-host1** and **ccf-host2** on **Subnet-1**.

Perform the following:

1. Open a remote terminal console on your first DC-series VM as per previous.
2. Repeat above step for your second DC-series VM, as you will need to be remotely connected to both of them.
3. Now that you are logged in on the two VMs, first make sure that you're in your home directory on each of the VMs. Type the following command if not:

```
$ cd ~
```

4. It's now time to download CCF. To do so, on each of the VMs, type the following command:

```
$ git clone https://github.com/microsoft/CCF
```

5. Before building CCF, you will need to install a few dependencies. Fortunately, a Shell script is available inside the CCF project to do that for you.
    a. Simply type the following command:

```
$ cd CCF/getting_started/setup_vm
```

    b. and then, execute the script:

```
$ ./setup.sh
```

Don’t forget to execute the commands on both of your VMs.

6. The next step consists in building CCF from its source files.
    a. Go back into the root directory of CCF by typing:

```
$ cd ../..
```

    b. Create a *build* directory:

```
$ mkdir build
```

    c. Then, go inside it:

```
$ cd build
```

    d. You have two choices there: you can build CCF by enabling all log messages, or not. Enabling them can be useful to understand what is going on when the node is online, but it can be a bit overwhelming as a lot of log messages are displayed on the screen. Type one of the following commands:

```
$ cmake -GNinja ..
```

(without all logs)

-or-

```
$ cmake -DVERBOSE_LOGGING=1 -GNinja ..
```

(with all logs)

    e. In both of the cases, simply type the following command after that to build the solution:

```
$ ninja
```

Note To learn more about CCF installation (build arguments, etc.), see article GETTING STARTED[24] of the CCF documentation.

Congratulations! CCF is installed at this stage, you're now able to run it. There are multiple steps to do so:

1. Launch two instances of CCF: one on each VM. They will become nodes of the network.
2. Each node will then have a quote file (used for remote attestation) and a cert file (associated to its public key). As such, the second node (the one which will join the network later) need to send its quote and cert file to the first one (over a TLS connection).
3. The first node (i.e. the one which will launch the network) need to generate a file called *nodes.json*, which describes the initial structure of the network when launched for the first time. It contains nodes IPs, Raft & TLS ports, and quote and cert files (which takes the form of a binary array).
4. Based to the previous file, the first node must generate the genesis transaction, which will be the anchor point of the starting blockchain. This will also generate a json script used to launch the network.
5. Finally, the first node will launch the blockchain, and the second node will get a signal to automatically join it.

Note Manually doing this step is quite complicated, but if you want to deep dive into CCF, see article STARTING UP THE NETWORK[25] of the CCF documentation.

In this guide, you will use instead a Python script. This script aims at easing the deployment of your test infrastructure, by executing each required command automatically, providing that you give suitable VMs information and credentials.

To get it, simply go on the CCF sample repository (available here[26]), then click on clone and copy the .git URL which appeared. Then, in your VMs terminals, type the following command and press ENTER:

```
$ git clone <.git URL>
```

The python script is located inside the newly created directory *ccf-samples/CCF install files*. Provided that you're inside the build directory, type the following command in your terminal console to copy the launching script from the repository to your current build directory:

```
$ cp "ccf-samples/CCF install files/network.py"
```

Then, run it by typing:

```
$ python3 network.py run
```

[24] GETTING STARTED: https://microsoft.github.io/CCF/getting_started.html

[25] STARTING UP THE NETWORK: https://microsoft.github.io/CCF/start_network.html

[26] Guide samples on the GitHub repo: https://aka.ms/CCDevGuideSamples

You will have to provide the following information:

- **IP address of the first node**. You can use the public VM IP or private VM IP (10.x.x.x) if you created a virtual network.
- **IP address of the second node**. (Same rules as the first node IP), username and password.
- **Raft and TLS ports**. They will be used by your nodes to communicate, e.g. 7465, 47585, etc.

Important note The script will not work if your CCF directory isn't located in *~/CCF/*, but it should be at the right place if you correctly followed the guide.

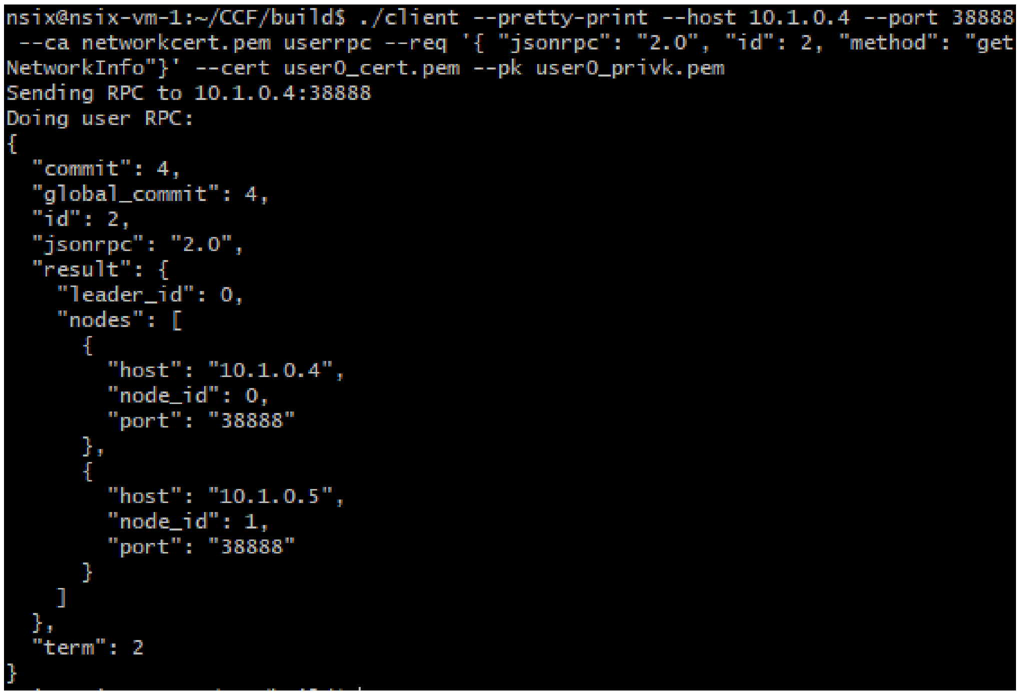
If your network is online, you should be able to run a JSON-RPC request to get online nodes' list.

Perform the following steps:

1. Open a remote terminal console on your first DC-series VM as per section § Connecting to your nodes.
2. Type the following command:

```
$ ./client –pretty-print –host <your node IP> --port <your node TLS port> --ca networkcert.pem
userrpc --req '{ "jsonrpc": "2.0", "id": 2, "method": "getNetworkInfo"}' --cert user0_cert.pem --pk
user0_privk.pem
```

Hereafter is an example of the expected result; you can see that 2 nodes are participating to the network.



```
nsix@nsix-vm-1:~/CCF/build$ ./client --pretty-print --host 10.1.0.4 --port 38888
 --ca networkcert.pem userrpc --req '{ "jsonrpc": "2.0", "id": 2, "method": "get
NetworkInfo"}' --cert user0_cert.pem --pk user0_privk.pem
Sending RPC to 10.1.0.4:38888
Doing user RPC:
{
  "commit": 4,
  "global_commit": 4,
  "id": 2,
  "jsonrpc": "2.0",
  "result": {
    "leader_id": 0,
    "nodes": [
      {
        "host": "10.1.0.4",
        "node_id": 0,
        "port": "38888"
      },
      {
        "host": "10.1.0.5",
        "node_id": 1,
        "port": "38888"
      }
    ]
  },
  "term": 2
}
```

Also, this is an example of a JSON-RPC request. This protocol is used with CCF to perform operations over the network.

This concludes this module.

In the next module, you're going to dive into all those operations to understand (some of) the CCF capabilities.

Module 3: Sending requests to the CCF network

You will see in this module how to start and leverage the network between your newly created nodes.

Now that your CCF network is online, you are indeed able to communicate with it, to send transactions or requests over a JSON-RPC endpoint, which takes as an input a JSON message and return another JSON output.

If you want the complete list of allowed methods, feel free to read the JSON-RPC documentation here[27].

To send a request to a node, execute the following command from the terminal console:

```
$ ./client –pretty-print –host <your node IP> --port <your node TLS port> --ca networkcert.pem
userrpc --req '{ "jsonrpc": "2.0", "id": 2, "method": "<method name>", “params”: {<method
parameters>}}' --cert user0_cert.pem --pk user0_privk.pem
```

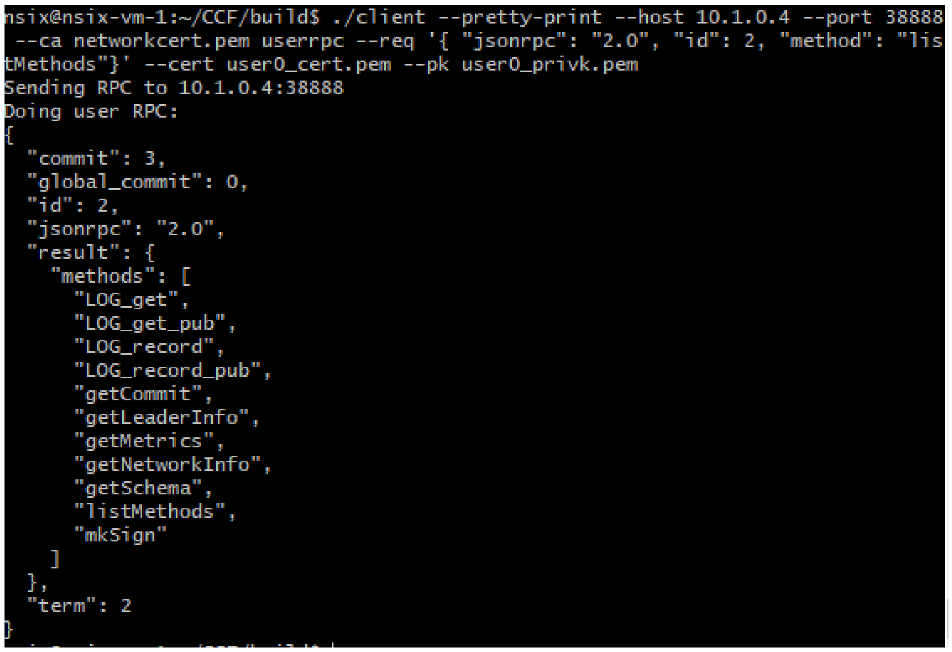
Please note that `params` is optional, it depends of the executed method.

For some commands, like *User management* methods, you will directly use some programs built at the same time as CCF. We’re going to came across those executables in this section.

Common methods

To start with these commands, we’re going to execute the method `listMethods` to get in return the list of all allowed methods. As you launched our network over the CCF Logging sample application, you will see that four methods exists to handle this application:

[27] RPC API: https://microsoft.github.io/CCF/rpc_api.html



```
nsix@nsix-vm-1:~/CCF/build$ ./client --pretty-print --host 10.1.0.4 --port 38888
 --ca networkcert.pem userrpc --req '{ "jsonrpc": "2.0", "id": 2, "method": "lis
tMethods"}' --cert user0_cert.pem --pk user0_privk.pem
Sending RPC to 10.1.0.4:38888
Doing user RPC:
{
  "commit": 3,
  "global_commit": 0,
  "id": 2,
  "jsonrpc": "2.0",
  "result": {
    "methods": [
      "LOG_get",
      "LOG_get_pub",
      "LOG_record",
      "LOG_record_pub",
      "getCommit",
      "getLeaderInfo",
      "getMetrics",
      "getNetworkInfo",
      "getSchema",
      "listMethods",
      "mkSign"
    ]
  },
  "term": 2
}
```

We're going to use them later, in a dedicated part. For the moment, let's see interesting common methods. One of them is `getLeaderInfo`.

CCF is based on an algorithm called Raft[28], which is in charge of electing a Leader between all nodes in the network. The Leader is the only node allowed to process transaction inside the network, as other nodes are called Followers and are waiting for orders from the Leader node.

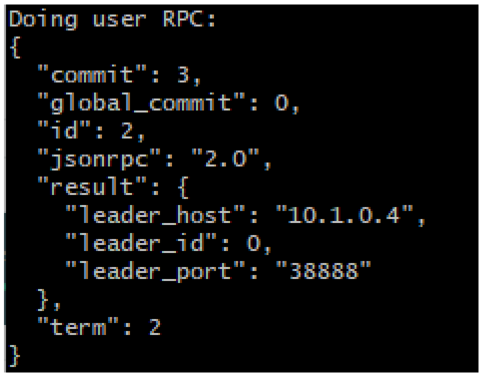
This guarantee that there is a consensus between all nodes, as they only add transactions validated by the Leader.

In your context, your know that our Leader is vm-1 since you've used it to start the network, but sometimes it could be useful to learn how to request information about the leader from any node of the network, to do some transactions later.

Let's execute this command on one of our nodes, to see the result:

```
$ ./client --pretty-print --host 10.1.0.4 --port 38888
 --ca networkcert.pem userrpc --req '{ "jsonrpc": "2.0", "id": 2, "method": "get
LeaderInfo"}' --cert user0_cert.pem --pk user0_privk.pem
```

[28] Raft (computer science): https://en.wikipedia.org/wiki/Raft_(computer_science)



```
Doing user RPC:
{
  "commit": 3,
  "global_commit": 0,
  "id": 2,
  "jsonrpc": "2.0",
  "result": {
    "leader_host": "10.1.0.4",
    "leader_id": 0,
    "leader_port": "38888"
  },
  "term": 2
}
```

As you can see, we're getting back his IP and TLS port, which is enough to send him transaction.

User management methods

When someone wants to administrate members and users, member methods should be called. To illustrate this part, you're going to add a user. First, your need to generate him/her his/her cert and his/her private key:

```
$ ./genesisgenerator cert –name "newmember"
```

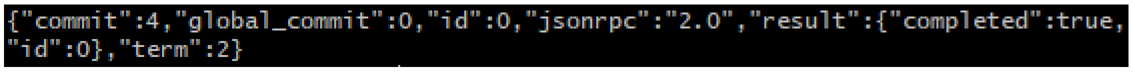
You should see those new files when doing ls inside your directory.

Adding a member in the network is not like another request. This type of request must be accepted by the majority of the members (aka reach the quorum).

To do so, a proposal must be submitted to the network, using `memberclient` executable. Type the following command:

```
$ ./memberclient add_member --ca=networkcert.pem --member_cert=<New member cert> --cert=<Already
accepted member cert> --privk=<Already accepted member private key> --host=<Node IP> --port=<TLS
port>
```

If it worked, you should see the following message, indicating an id which is the proposal id:



```
{"commit":4,"global_commit":0,"id":0,"jsonrpc":"2.0","result":{"completed":true,
"id":0},"term":2}
```

Therefore, members can choose between accepting or denying access to the network to the new member.

This is also possible to add new nodes from this executable. To do that, call `memberclient` executable with the add_node command, provide as a parameter a JSON object containing information about nodes to be added (exactly like the file *nodes.json*), but also your cert/privk and node IP/port like examples above:

```
$ ./memberclient add_node --nodes_to_add node.json --ca=networkcert.pem --cert=<Already accepted
member cert> --privk=<Already accepted member private key> --host=<Node IP> --port=<TLS port>
```

The process is the same as members, you will get a proposal id and you can accept or reject it with the `accept_node` and `retire_node` commands.

Logging application methods

As you saw before, the Logging application of our network is constituted with 4 methods:

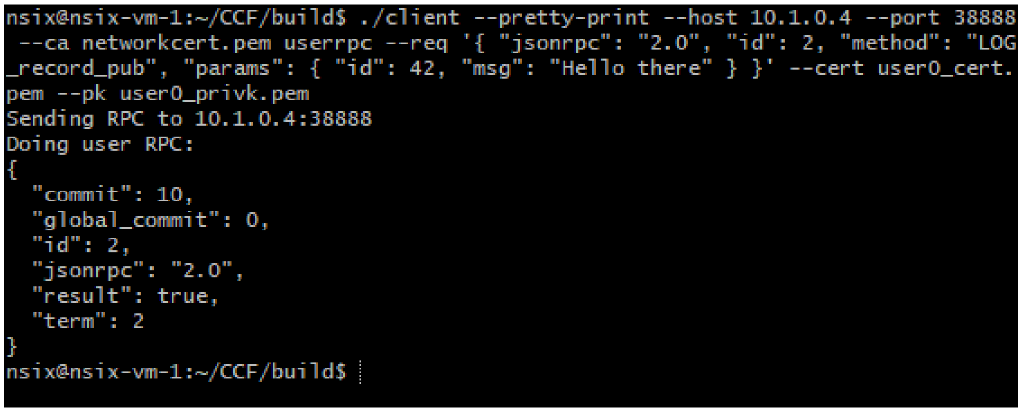
1. `LOG_record`. Post a private log (a small text message) on the blockchain at a given index. As it is a private log, it will be encrypted on the distributed ledger and therefore only readable by nodes on the network.
2. `LOG_record_pub`. Perform the same action as `LOG_record` but will not be encrypted after sending it. Anybody which can open the ledger file will see the log in plain text.
3. `LOG_get`. Return a log from its index.
4. `LOG_get_pub`. Return a log from its id, only works if the log is public.

Let's try to send a transaction which posts a new log:

```
$ ./client –pretty-print –host <your node IP> --port <your node TLS port> --ca networkcert.pem
userrpc --req '{ "jsonrpc": "2.0", "id": 2, "method": "LOG_record_pub", "params": { "id": 42, "msg":
"Hello there"}}' --cert user0_cert.pem --pk user0_privk.pem
```

You can see that the method name is `LOG_record_pub`, and parameters are the newly created log index, and log content.

Important note The host pointed with that command must be the leader, as this is the only node which can process transactions. If you try to send this transaction to a follower node, you will get an error in return.

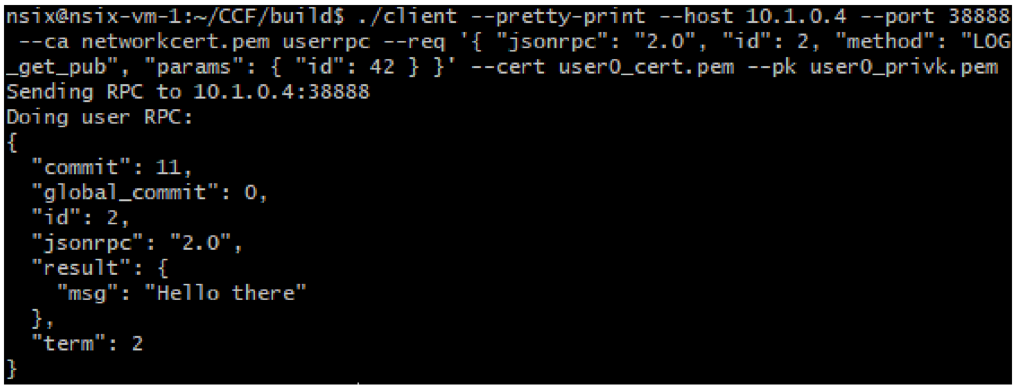


```
nsix@nsix-vm-1:~/CCF/build$ ./client --pretty-print --host 10.1.0.4 --port 38888
 --ca networkcert.pem userrpc --req '{ "jsonrpc": "2.0", "id": 2, "method": "LOG
_record_pub", "params": { "id": 42, "msg": "Hello there" } }' --cert user0_cert.
pem --pk user0_privk.pem
Sending RPC to 10.1.0.4:38888
Doing user RPC:
{
  "commit": 10,
  "global_commit": 0,
  "id": 2,
  "jsonrpc": "2.0",
  "result": true,
  "term": 2
}
nsix@nsix-vm-1:~/CCF/build$
```

If everything went well, you should get this JSON in return. Let's execute the get command to see if you can get back your log:

```
$ ./client –pretty-print –host <your node IP> --port <your node TLS port> --ca networkcert.pem
userrpc --req '{ "jsonrpc": "2.0", "id": 2, "method": "LOG_get_pub", "params": { "id": 42 }}' --cert
user0_cert.pem --pk user0_privk.pem
```

Following is the result:



```
nsix@nsix-vm-1:~/CCF/build$ ./client --pretty-print --host 10.1.0.4 --port 38888
 --ca networkcert.pem userrpc --req '{ "jsonrpc": "2.0", "id": 2, "method": "LOG
_get_pub", "params": { "id": 42 } }' --cert user0_cert.pem --pk user0_privk.pem
Sending RPC to 10.1.0.4:38888
Doing user RPC:
{
  "commit": 11,
  "global_commit": 0,
  "id": 2,
  "jsonrpc": "2.0",
  "result": {
    "msg": "Hello there"
  },
  "term": 2
}
```

You can see that in the `result` field, our message appeared. You can also have a look at the `commit` field: this id keeps track of transactions done with the network. Each time you send something to it, this it is incremented by 1.

This both concludes this module and this mini guide for developers.

Appendix. Support files for the guide

network.py file

```
import os
import json
import datetime
import getpass
import paramiko
import sys

DEBUG = True

def log_text(text):
    if(DEBUG):
        print(str(datetime.datetime.now()) + ' : ' + str(text))

def cert_bytes(cert_file_name):
    """
    Parses a pem certificate file into raw bytes and appends null character.
    """
    with open(cert_file_name, "rb") as pem:
        chars = []
        for c in pem.read():
            chars.append(ord(c))
        # mbedtls demands null-terminated certs
        chars.append(0)
        return chars


def quote_bytes(quote_file_name):
    """
    Parses a binary quote file into raw bytes.
    """
    with open(quote_file_name, "rb") as quote:
        chars = []
        for c in quote.read():
            chars.append(ord(c))
        return chars

def reset_workspace():
    os.system("sudo pkill cchost")
    os.system("rm -rf tx0* gov* *.pem quote* nodes.json startNetwork.json joinNetwork.json ledger
parsed_* sealed_*")

def reset_remote_workspace(c):
    c.exec_command("sudo pkill cchost")
    c.exec_command("cd ~/CCF/build && rm -rf tx0* gov* *.pem quote* nodes.json startNetwork.json
joinNetwork.json ledger parsed_* sealed_*")

def generate_members_certs(nb_members_certs, nb_users_certs):
    for x in range(0, nb_members_certs):
        os.system("./genesisgenerator cert --name=./member" + str(x))
    for x in range(0, nb_users_certs):
        os.system("./genesisgenerator cert --name=./user" + str(x))
```

```
def generate_members_certs_on_remote(c, nb_members_certs, nb_users_certs):
    for x in range(0, nb_members_certs):
        c.exec_command("cd ~/CCF/build && ./genesisgenerator cert --name=member" + str(x))
    for x in range(0, nb_users_certs):
        c.exec_command("cd ~/CCF/build && ./genesisgenerator cert --name=user" + str(x))

def generate_nodes_json(info):
    with open('./nodes.json', 'w') as outfile:
        json.dump(
        [
            {
                "host": info["node_address_1"],
                "raftport": info["raft_port"],
                "pubhost": info["node_address_1"],
                "tlsport": info["tls_port"],
                "cert": cert_bytes("./0.pem"),
                "quote": quote_bytes("./quote0.bin"),
                "status": 0,
            },
            {
                "host": info["node_address_2"],
                "raftport": info["raft_port"],
                "pubhost": info["node_address_2"],
                "tlsport": info["tls_port"],
                "cert": cert_bytes("./1.pem"),
                "quote": quote_bytes("./quote1.bin"),
                "status": 0,
            },
        ]
        , outfile)

def start_remote_node(info, c):
    reset_remote_workspace(c)
    c.exec_command("cd ./CCF/build && ./cchost "
        "--enclave-file=./libloggingenc.so.signed "
        "--raft-election-timeout-ms=100000 "
        "--raft-host=" + info["node_address_2"] + " "
        "--raft-port=" + info["raft_port"] + " "
        "--tls-host=" + info["node_address_2"] + " "
        "--tls-pubhost=" + info["node_address_2"] + " "
        "--tls-port=" + info["tls_port"] + " "
        "--ledger-file=./ledger "
        "--node-cert-file=./1.pem "
        "--enclave-type=debug "
        "--log-level=info "
        "--quote-file=./quote1.bin &")

def start_node(info):
    os.system("./cchost "
        "--enclave-file=./libloggingenc.so.signed "
        "--raft-election-timeout-ms=100000 "
        "--raft-host=" + info["node_address_1"] + " "
        "--raft-port=" + info["raft_port"] + " "
        "--tls-host=" + info["node_address_1"] + " "
        "--tls-pubhost=" + info["node_address_1"] + " "
        "--tls-port=" + info["tls_port"] + " "
        "--ledger-file=./ledger "
        "--node-cert-file=./0.pem "
        "--enclave-type=debug "
        "--log-level=info "
        "--quote-file=./quote0.bin & ")
```

```
def retrieve_remote_node_certs(c):
    try:
        sftp = c.open_sftp()
        sftp.get("./CCF/build/quote1.bin", "./quote1.bin")
        sftp.get("./CCF/build/1.pem","./1.pem")
        return True
    except:
        print("Error: failed to retrieve remote node certs. Check node health, then retry.")
        return False

def send_network_info_to_remote_node(c):
    try:
        sftp = c.open_sftp()
        sftp.put("./networkcert.pem", "./CCF/build/networkcert.pem")
        return True
    except:
        print("Error: failed to send network cert. Check node health, then retry.")
        return False

def connect_remote_node(info):
    try:
        c = paramiko.SSHClient()
        c.load_system_host_keys()
        c.set_missing_host_key_policy(paramiko.WarningPolicy)
        c.connect(info["node_address_2"], port=22, username=info["node_user_2"],
password=info["node_pwd_2"])
        return c
    except:
        print("Error: unable to connect to remote node. Check node health and credentials, then
retry.")
        return False

def get_light_node_info():
    node_address_2 = str(raw_input("Remote node IP address: "))
    node_user_2 = str(raw_input("Remote node IP username: "))
    node_pwd_2 = getpass.getpass()

    return {
        "node_address_2": node_address_2,
        "node_user_2": node_user_2,
        "node_pwd_2": node_pwd_2,
    }

def get_node_info():
    node_address_1 = str(raw_input("Local node IP address: "))
    node_address_2 = str(raw_input("Remote node IP address: "))
    node_user_2 = str(raw_input("Remote node IP username: "))
    node_pwd_2 = getpass.getpass()
    raft_port = str(raw_input("Raft port: "))
    tls_port = str(raw_input("TLS port: "))

    return {
        "node_address_1": node_address_1,
        "node_address_2": node_address_2,
        "node_user_2": node_user_2,
        "node_pwd_2": node_pwd_2,
        "raft_port": raft_port,
        "tls_port": tls_port,
    }
```

```
def reset_workspaces():
    info = get_light_node_info()
    log_text("Cleaning older files and resetting server on local node ...")
    reset_workspace()
    log_text("Done.")

    log_text("Connecting to remote node ...")
    c = connect_remote_node(info)
    log_text("Done.")

    log_text("Cleaning older files and resetting server on remote node ...")
    reset_remote_workspace(c)
    log_text("Done.")

    c.close()

def run(args=None):
    info = get_node_info()
    log_text("Cleaning older files and resetting server")
    reset_workspace()

    log_text("Starting local node ...")
    start_node(info)

    log_text("Connecting to remote node ...")
    c = connect_remote_node(info)

    if(c):
        log_text("Starting remote node ...")
        start_remote_node(info, c)

        log_text("Waiting for nodes to start ...")
        os.system("sleep 8")

        log_text("Retrieving remote node certs ...")
        res = retrieve_remote_node_certs(c)

        if(res):
            log_text("Generating nodes.json file ...")
            generate_nodes_json(info)
            log_text("Done.")

            log_text("Generating members and users certs on local node ...")
            generate_members_certs(1,1)
            log_text("Done.")

            log_text("Generating members and users certs on remote node ...")
            generate_members_certs_on_remote(c,1,1)
            log_text("Done.")

            log_text("Generating genesis transaction ...")
            os.system("./genesisgenerator tx --gov-script=../src/runtime_config/gov.lua")

            log_text("Starting blockchain ...")
            os.system("./client --host=" + info["node_address_1"] + " --port=" + info["tls_port"] + "
--ca=./0.pem startnetwork --req=@startNetwork.json")

            log_text("Sending network cert to remote node ...")
            send_network_info_to_remote_node(c)
            log_text("Done.")
            log_text("Connecting remote node to blockchain ...")
            c.exec_command("cd ~/CCF/build && ./genesisgenerator joinrpc --network-
cert=./networkcert.pem --host=" + info["node_address_1"] + " --port=" + info["tls_port"])
            #print("cd ~/CCF/build && ./genesisgenerator joinrpc --network-cert=./1.pem --host=" +
info["node_address_1"] + " --port=" + info["tls_port"])
            c.exec_command("cd ~/CCF/build && ./client --host=" + info["node_address_2"] + " --port="
+ info["tls_port"] + " --ca=./1.pem joinnetwork --req=joinNetwork.json")
            #print("./client --host=" + info["node_address_2"] + " --port=" + info["tls_port"] + " --
ca=./1.pem joinnetwork --req=joinNetwork.json")
            log_text("Done.")

            log_text("Network online, setup complete.")
        c.close()
    else:
        os.system("sudo pkill cchost")
        os.system("cd ~/CCF/build  && rm -rf tx0* gov* *.pem quote* nodes.json startNetwork.json
joinNetwork.json 0 parsed_* sealed_*")

if __name__ == "__main__":
    if sys.argv[1] == "run":
        run()
    elif sys.argv[1] == "clean":
        reset_workspaces()
    else:
        print("How to use the script : ")
        print("-> python network.py run : launch the sample network")
        print("-> python network.py clean : clean all files generated from previous tests, on VMs
used")
```