# CÉCI HPC Training

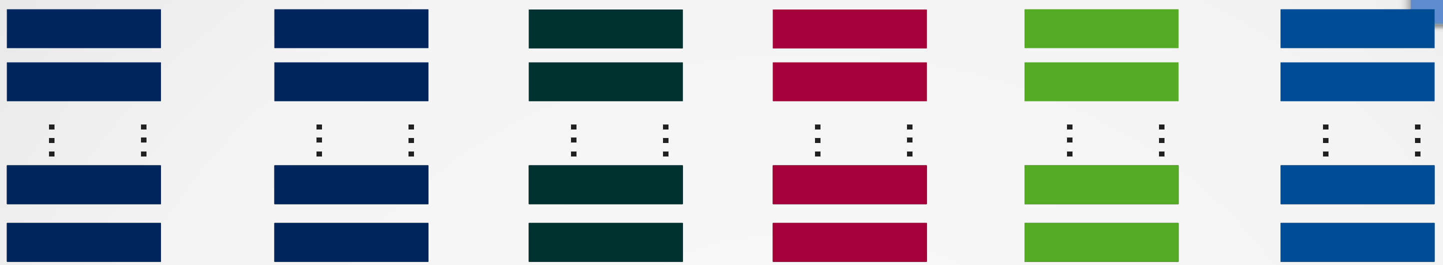
## Connecting with SSH from Linux or Mac: Introduction and advanced topics

Juan.Cabrera@unamur.be

# Connecting with Secure SHell

- SSH context
- SSH introduction
- Getting your key
- SSH client usage and configuration
- SSH frequent mistakes
- SSH Agents, Passphrase managers
- Proxies and (pseudo-)VPNs (shuttle)
- SSH-based file transfer (SCP, rsync, SSHFS)

# SSH context: CÉCI infrastructure

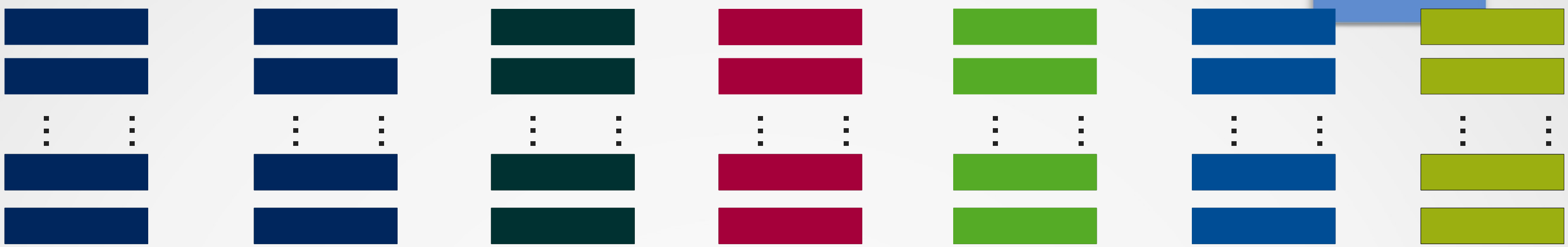


UCL Université catholique de Louvain · LIÈGE université · UMONS Université de Mons · UNIVERSITÉ DE NAMUR · ULB

CÉCI is:

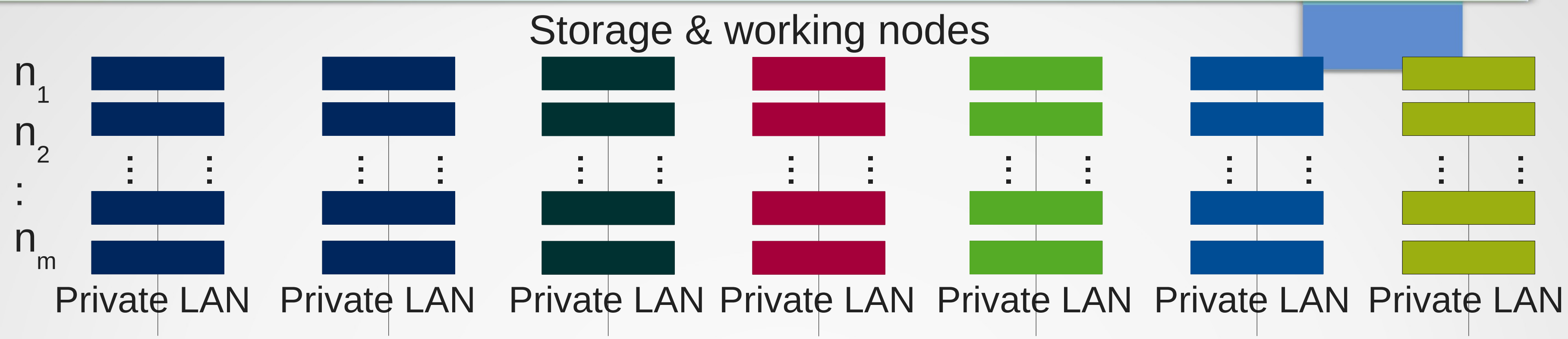
**6 computers clusters** from 5 French-speaking universities

# SSH context: CÉCI infrastructure



**Tier-1 facility** access for CÉCI user under special conditions

# SSH context: CÉCI infrastructure

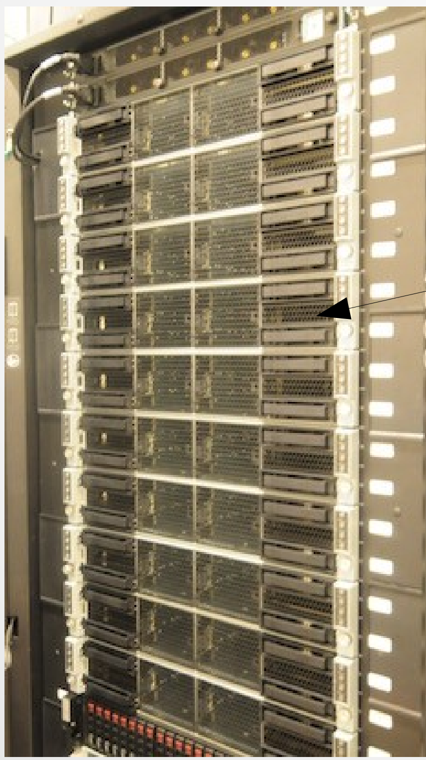



On each cluster
**Storage & working nodes** are interconnected in a private network

# SSH context: CÉCI infrastructure

- Example

Lemaitre3 (UCLouvain)

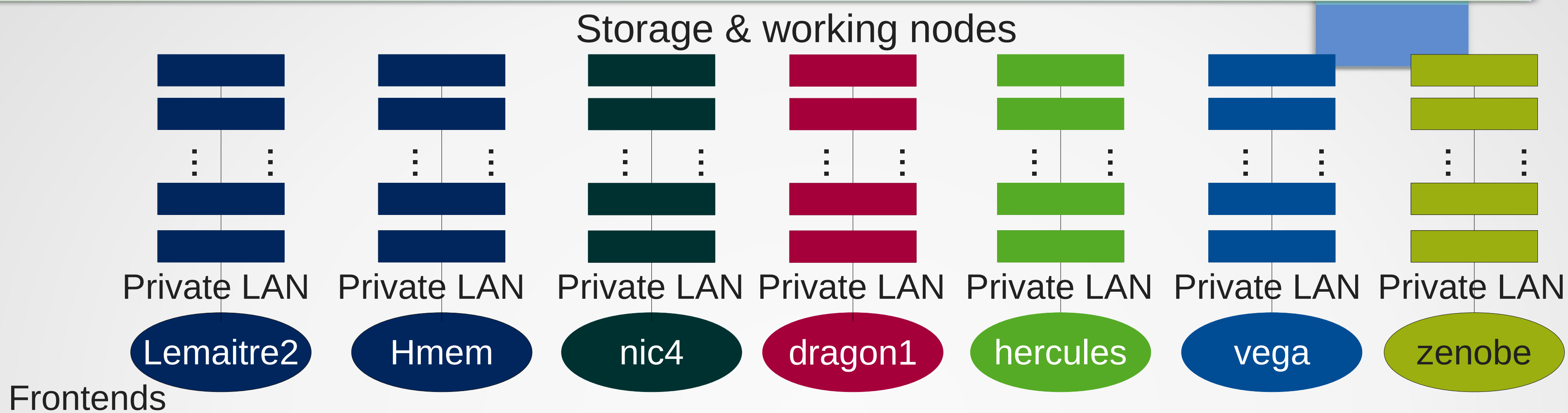


Nic4 (ULiege)

Working nodes

Interconnections

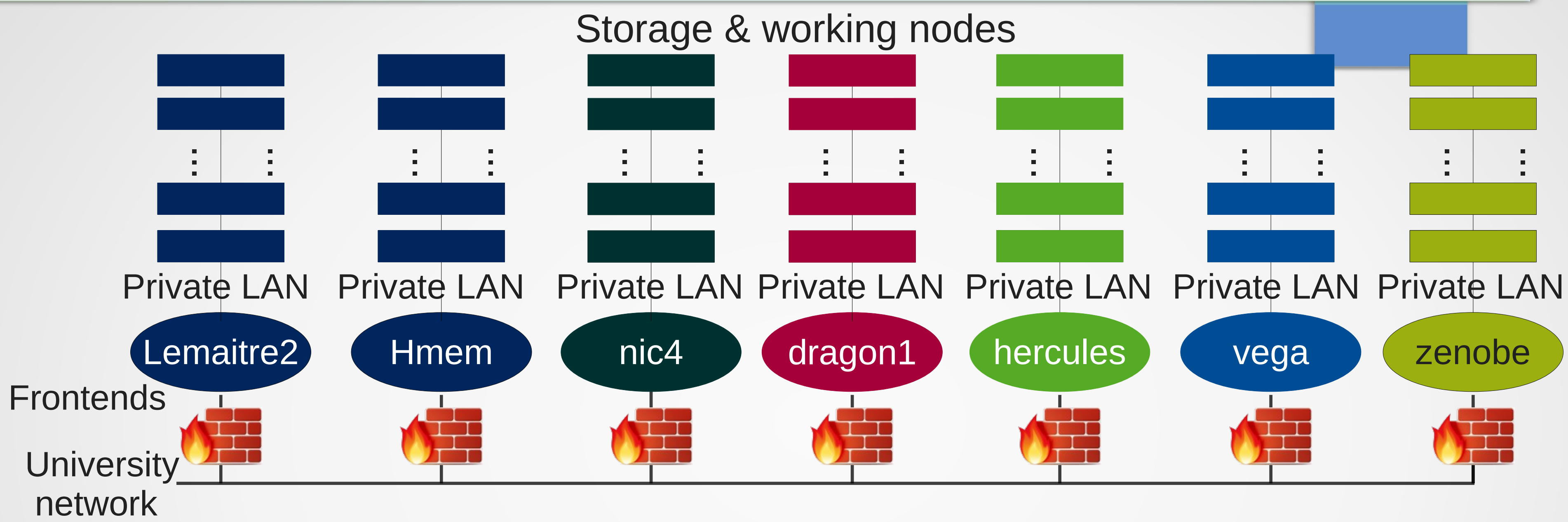
# SSH context: CÉCI infrastructure




User must connect to the **frontend** to

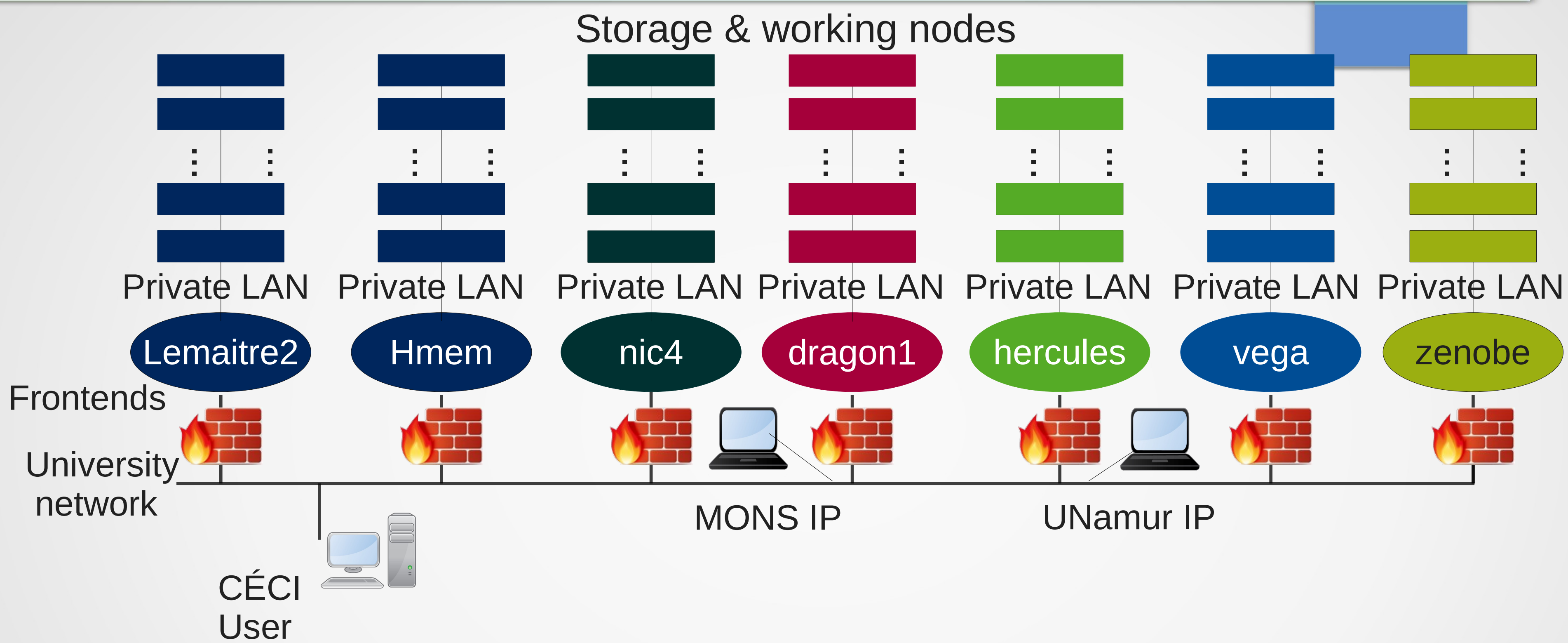
- **access** its storage data and **edit** files
- **submit jobs** to the working nodes
- **compiling** and **debugging**
- **transfer** data
- **Do not run heavy jobs** in the frontend

# SSH context: CÉCI infrastructure



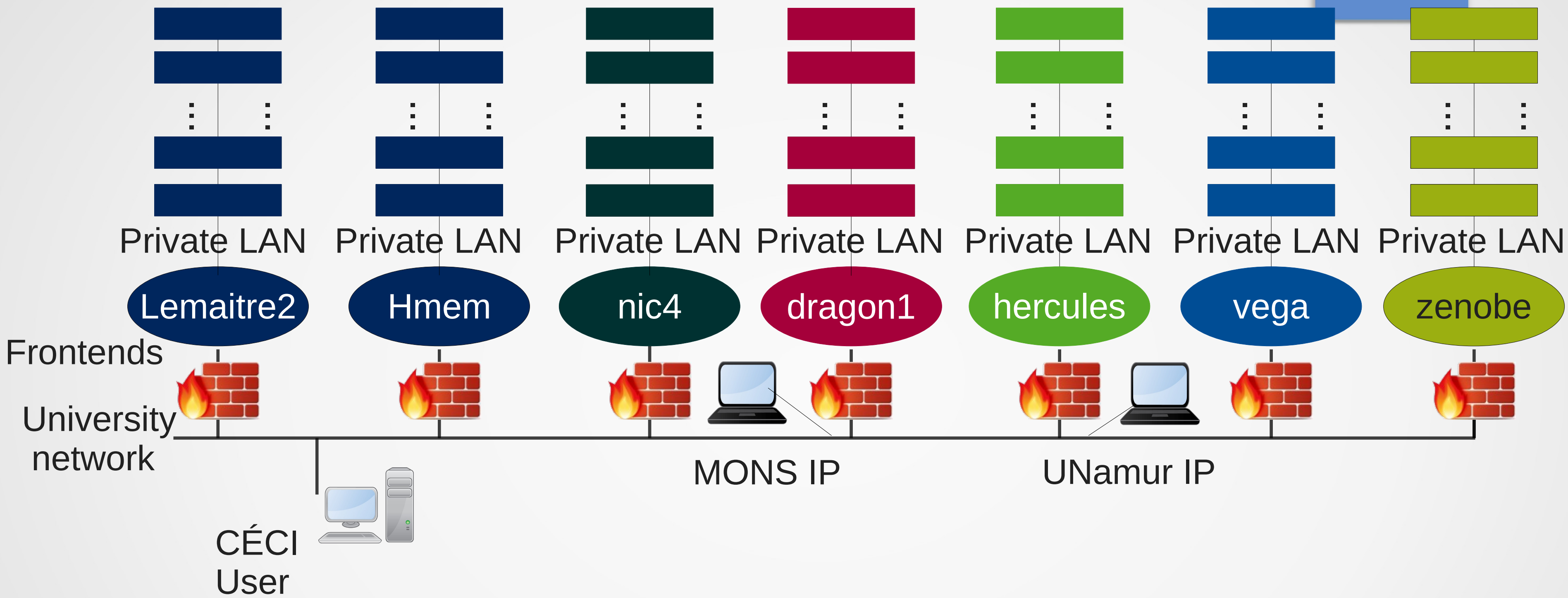

**frontends** access is protected by firewall rules

# SSH context: CÉCI infrastructure



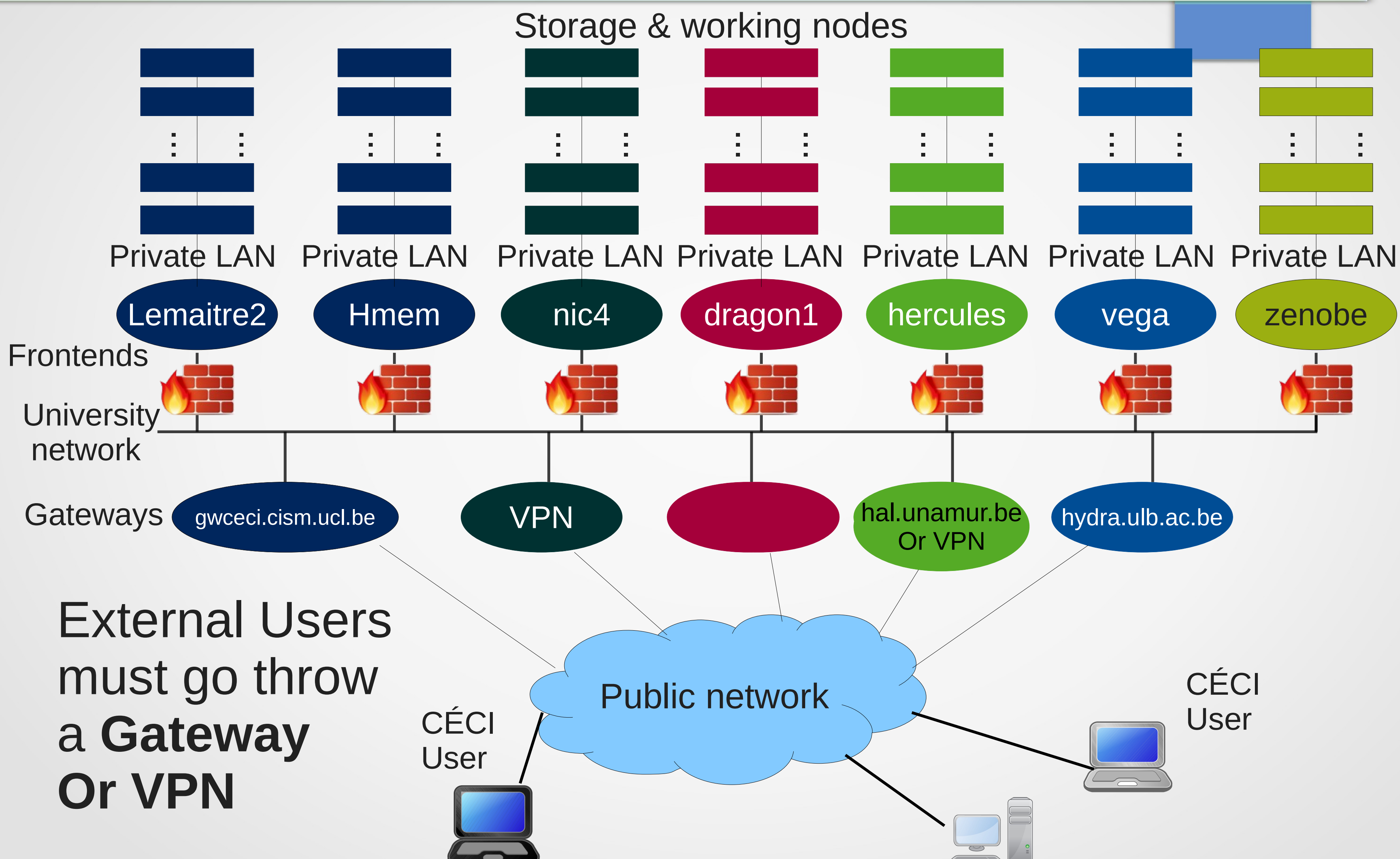

With the Firewall rules,
we can approximate the connections
by logical **private university network**

# SSH context: CÉCI infrastructure




Connections to frontends done via **SSH**
From a CÉCI university network

# SSH context: CÉCI infrastructure




External Users must go throw a **Gateway Or VPN**

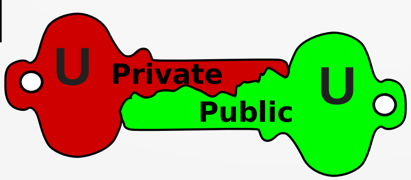
# SSH introduction: Public-Private key

An SSH identity uses **asymmetric cryptography** with **a pair of keys**, one private and one public

When you ask for a new or renew a CÉCI account at https://login.ceci-hpc.be,
2 keys are generated using your passphrase

`id_rsa.ceci`

Private



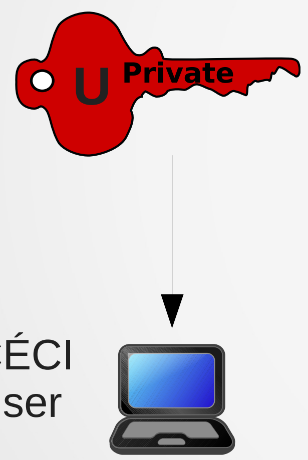

Public

`id_rsa.ceci.pub`

# SSH introduction: Public-Private key

The private key is **encrypted by the passphrase** and **sent by mail** to the user.

**It must be stored in a safe place in your computer.**




CÉCI User

```
$ cat id_rsa.ceci

-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,798194AFB2800B27

KnvjN+KM4NogUADgdVI7GawGEmxJtXl2NKbezDyI8aeUAYxHemgThcRMswe2DAPs
fCeAJkTZ/B23uAWRppVvuPwJtp/AD3cvYxY5jBvSwVlAUdrfOJauegGc99CqvDEV
...
...
wT/yGuuRi9xfn6/yY7wTDxeaJg5WRd54oq0jbpTPUQmZWjJ1cuzBNiioNBXAFTGD
OJkZChE7fLD+C7kvYH0J6u4NiXUWqVheNerl0OnCZuM770gY5P0Q7w==
-----END RSA PRIVATE KEY-----
```

For security reasons

**CÉCI does not keep a copy of the private key**.

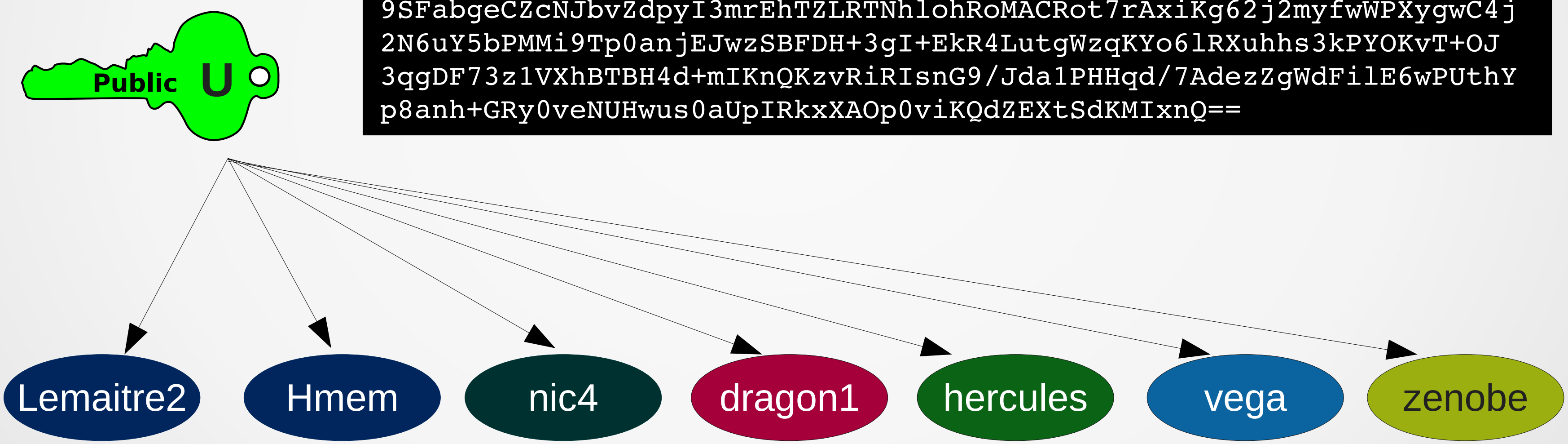
If you loose It, forget the passphrase or think it is compromised you must **retrieve a new key** at https://login.ceci-hpc.be

# SSH introduction: Public-Private key

Public key is **placed in frontends** for authentication.

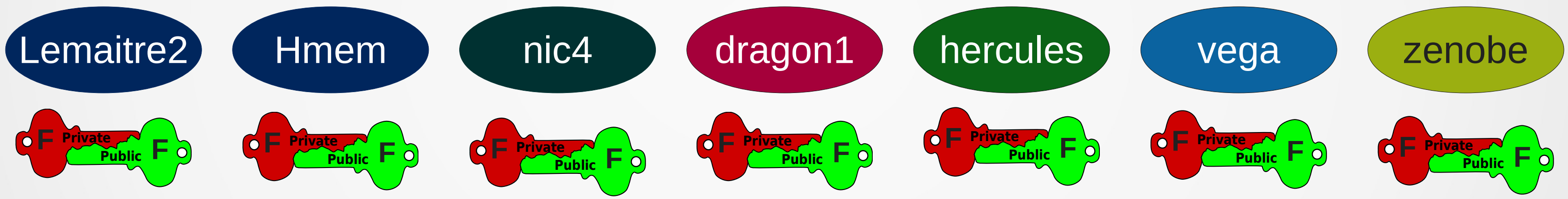
```
$ cat id_rsa.ceci.pub

ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEA2U59janaM1uhC4R1yL4Iozlx4FvQ6a
Q0tqIv9c6EHGj2wafVG8bxR1StYYecQ1oaY2C3AUeu9bTjtH9Rj5IPlvFf4OPAFMgU5
9SFabgeCZcNJbvZdpyI3mrEhTZLRTNhlohRoMACRot7rAxiKg62j2myfwWPXygwC4j
2N6uY5bPMMi9Tp0anjEJwzSBFDH+3gI+EkR4LutgWzqKYo6lRXuhhs3kPYOKvT+OJ
3qgDF73z1VXhBTBH4d+mIKnQKzvRiRIsnG9/Jda1PHHqd/7AdezZgWdFilE6wPUthY
p8anh+GRy0veNUHwus0aUpIRkxXAOp0viKQdZEXtSdKMIxnQ==
```



# SSH introduction: Public-Private key

Each frontend as it's own private and public key



# SSH introduction: protocol

The SSH  connection and authentication protocol has 5 main phases

# SSH introduction: protocol

1) Establish TCP Connection to frontend
2) Identification string Exchange (check if good ssh version)
3) Algorithm negotiation (which encryption algorithm is used)
4) Diffie-Hellman Key Exchange (User gets frontend's public key)
5) User Authentication and Authorization
   (User send his/her login and public key)

# Getting your private key

Users without email account access, without CÉCI university email or who does not need a CÉCI account can use a key for one of the guest accounts.

http://www.cism.ucl.ac.be/Services/Formations/pk/

Save the private key in a file named id_rsa.ceci

# Getting your private key

Users with email account access can ask for an account at:

https://login.ceci-hpc.be/init/

- Click 'Create Account'
- Type in your email address
- Click on the link sent to you by email.
- Fill-in the form and hit the “Submit” button.
- Wait ... (A sysadmin is reviewing your information).
- receive your private key by email.
- Store the **id_rsa.ceci** file in a safe location.

# SSH client : **Linux & MacOS**

SSH client for connection is already installed

# SSH client usage

C.E.C.I

1) Save your key id_rsa.ceci file from your e-mail to your home directory
2) Open a terminal
3) Create the .ssh directory if it does not exist

```
$ mkdir ~/.ssh
```

4) Move your key to this directory

```
$ mv id_rsa.ceci  ~/.ssh/.
```

5) Change the permissions of the file so that only you can read it

```
$ chmod 600 ~/.ssh/id_rsa.ceci
```

6) Check the permissions. The follow command :

```
$ ls -l ~/.ssh/id_rsa.ceci
```

Must output -rw------- permissions

7) Now you can connect to a CÉCI cluster, e.g. Hmem, with

```
$ ssh -i ~/.ssh/id_rsa.ceci yourlogin@hmem.cism.ucl.ac.be
```

# SSH client usage

## Example

```
$ ssh -i ~/.ssh/id_rsa.ceci jcabrera@hmem.cism.ucl.ac.be
```

# SSH client usage

## Example

```
$ ssh -i ~/.ssh/id_rsa.ceci jcabrera@hmem.cism.ucl.ac.be
The authenticity of host 'hmem.cism.ucl.ac.be (130.104.1.220)' can't be established.
RSA key fingerprint is 06:54:39:a0:5c:b5:56:b3:29:9e:96:67:a0:4a:c1:ff.
Are you sure you want to continue connecting (yes/no)?
```

FIRST TIME you connect to a frontend from a client, you will be asked to accept the Public Key

Check the key fingerprint from CÉCI web site

http://www.ceci-hpc.be/clusters.html#hmem

SUPPORT: egs-cism@listes.uclouvain.be

Server SSH key fingerprint: (What's this?)

MD5: 06:54:39:a0:5c:b5:56:b3:29:9e:96:67:a0:4a:c1:ff

SHA256:

Xi4r0aNViNgg9KjnENiUFkEWPwnJGAjbknlX+m7Clm0

# SSH client usage

## Example

```
$ ssh -i ~/.ssh/id_rsa.ceci jcabrera@hmem.cism.ucl.ac.be
The authenticity of host 'hmem.cism.ucl.ac.be (130.104.1.220)' can't be established.
RSA key fingerprint is 06:54:39:a0:5c:b5:56:b3:29:9e:96:67:a0:4a:c1:ff.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'hmem.cism.ucl.ac.be' (RSA) to the list of known hosts.
Enter passphrase for key '/home/jcabrera/.ssh/id_rsa.ceci':
```

Now, the hmem public key is stored in your **know_host** file

Enter the **passphrase** you set when you create the account
This will decrypt your private key

# SSH client usage

## Example

```
$ ssh -i ~/.ssh/id_rsa.ceci jcabrera@hmem.cism.ucl.ac.be
The authenticity of host 'hmem.cism.ucl.ac.be (130.104.1.220)' can't be established.
RSA key fingerprint is 06:54:39:a0:5c:b5:56:b3:29:9e:96:67:a0:4a:c1:ff.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'hmem.cism.ucl.ac.be' (RSA) to the list of known hosts.
Enter passphrase for key '/home/jcabrera/.ssh/id_rsa.ceci':
Welcome to
          __  __     __    __     ______     __    __
         /\ \_\ \   /\ "-./  \   /\  ___\   /\ "-./  \
         \ \  __ \  \ \ \-./\ \  \ \  __\   \ \ \-./\ \
          \ \_\ \_\  \ \_\ \ \_\  \ \_____\  \ \_\ \ \_\
           \/_/\/_/   \/_/  \/_/   \/_____/   \/_/  \/_/
                   HighMemory CISM-CECI cluster
…
…
…
   Don't know where to start?
            --> http://www.ceci-hpc.be/install_software.html
            --> http://www.ceci-hpc.be/slurm_tutorial.html
[jcabrera@hmem00 ~]$
```

You are now connected !!

# SSH client usage: **Frequent mistakes**

## The permissions on your key file are not correct

If, after running ssh hmem, for instance, you see something like:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@         WARNING: UNPROTECTED PRIVATE KEY FILE!          @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
Permissions 0644 for '/home/dfr/.ssh/id_rsa.ceci' are too open.
It is recommended that your private key files are NOT accessible by others.
This private key will be ignored.
bad permissions: ignore key: /home/dfr/.ssh/id_rsa.ceci
dfr@hmem.cism.ucl.ac.be's password:
it means that Permissions 0644 for '/home/dfr/.ssh/id_rsa.ceci' are too open.
Change them to 600 as explained in the first section of this document.
```

It means that
**Permissions** 0644 for '/home/dfr/.ssh/id_rsa.ceci' **are too open**.
Change them to 600 as explained previously

```
$ chmod 600 ~/.ssh/id_rsa.ceci
```

# SSH client usage: **Frequent mistakes**

## You did not specify the correct path to your SSH key

If, after running ssh, you are being asked for a password directly,

```
$ ssh hmem
dfr@hmem.cism.ucl.ac.be's password:
```

it means that your SSH client did not use the SSH key. Make sure you either used the -i option or that your .ssh/config is properly configured.

## You used a wrong username or tried to connect before your keys are synchronized

If, after running ssh, you are being asked for a passphrase, then a password,

```
$ ssh hmem
Enter passphrase for key '/home/dfr/.ssh/id_rsa.ceci':
dfr@hmem.cism.ucl.ac.be's password:
```

it often means that the user name you are using is not the correct one. It could also mean that you are trying to connect with the new private key while it has not been synchronized to the cluster yet (clusters are not synchronized simultaneously you need to **wait ~30 min**.)

# SSH client usage

You can use -v, -vv or -vvv to troubleshooting a session

Identification string Exchange

Algorithm negotiation

Diffie-Hellman Key Exchange

User Authentication and Authorization

```
$ ssh -v -i ~/.ssh/id_rsa.ceci yourlogin@hmem.cism.ucl.ac.be
…
debug1: Local version string SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2
debug1: Remote protocol version 2.0, remote software version OpenSSH_5.3
…
debug1: SSH2_MSG_KEXINIT sent
debug1: SSH2_MSG_KEXINIT received
…
The authenticity of host 'hmem.cism.ucl.ac.be (130.104.1.220)' can't be established.
RSA key fingerprint is 06:54:39:a0:5c:b5:56:b3:29:9e:96:67:a0:4a:c1:ff.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'hmem.cism.ucl.ac.be' (RSA) to the list of known hosts.
debug1: ssh_rsa_verify: signature correct    server authenticity
…
debug1: SSH2_MSG_NEWKEYS received    communication is encrypted with symmetric key
…
debug1: Offering RSA public key: /home/jcabrera/.ssh/id_rsa.ceci
debug1: Server accepts key: pkalg ssh-rsa blen 277
…
Enter passphrase for key '/home/jcabrera/.ssh/id_rsa.ceci':
…
debug1: Authentication succeeded (publickey).    user authenticity
```

# Exercise

Make your first connection to hmem.cism.ucl.ac.be

# SSH configuration

You can reduce the length of the follow command:

```
$ ssh -i ~/.ssh/id_rsa.ceci yourlogin@hmem.cism.ucl.ac.be
```

Edit or create the configuration file **~/.ssh/config** and add the contents generated by the following script:

http://www.ceci-hpc.be/sshconfig.html

# SSH configuration

```
# Generalities ---------------------------------
Host hmem lemaitre3 hercules dragon1 vega nic4
    ForwardAgent yes
    ForwardX11 yes
    IdentityFile ~/.ssh/id_rsa.ceci
# CÉCI clusters --------------------------------
Host hmem
    Hostname hmem.cism.ucl.ac.be
    User jsmith
Host lemaitre3 ...
Host hercules ...
Host dragon1 ...
Host vega ...
Host nic4 ...
```

**FowardX11** is needed to open any host program in the client display.

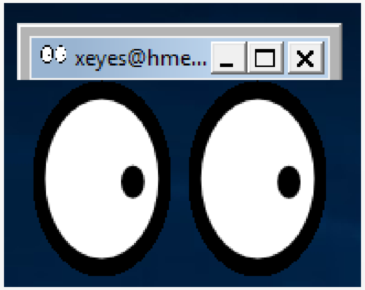
With **ForwardAgent** the connection to the agent is automatically forwarded to the remote side

Now you can connect with the command:

```
$ ssh hmem
```

# Exercise

- Create your configuration file
- Use the CECI Wizard to add all frontends
- And connect
- execute **xeyes** command on **hmem**



# SSH Agents, Passphrase managers

Use an SSH agent which will remember the passphrase so you do not have to type it in each time you issue the SSH command.

1) make sure you have an agent running

```
$ ssh-add -l
Could not open a connection to your authentication agent.
```

2) If you get "Could not open a connection to your authentication agent." start an agent with

```
$ eval $(ssh-agent)
```

3) add you key. Your key is decrypted and stored in memory

```
$ ssh-add ~/.ssh/id_rsa.ceci
Enter passphrase for /home/jcabrera/.ssh/id_rsa.ceci:
Identity added: /home/jcabrera/.ssh/id_rsa.ceci (/home/jcabrera/.ssh/id_rsa.ceci)
```

4) check the loaded key

```
$ ssh-add -l
2048 20:6c:8c:cd:e8:e6:9b:4f:8c:9c:d6:8a:eb:37:6d:17 /home/jcabrera/.ssh/id_rsa.ceci (RSA)
```

5) You can connect to the host without set the passphrase

```
$ ssh hmem
```

# SSH Agents, Passphrase managers

You can have an ssh-agent started automatically at login by using password managing software such as

Mac OS Keychain, KDE KWallet, Gnome Keyring (Seahorse), etc.

Gnome Keyring loads all private keys in ~/.ssh **which have the corresponding public key**.

You can generate the public key with the command

```
# ssh-keygen -y -f ~/.ssh/id_rsa.ceci > ~/.ssh/id_rsa.ceci.pub
```

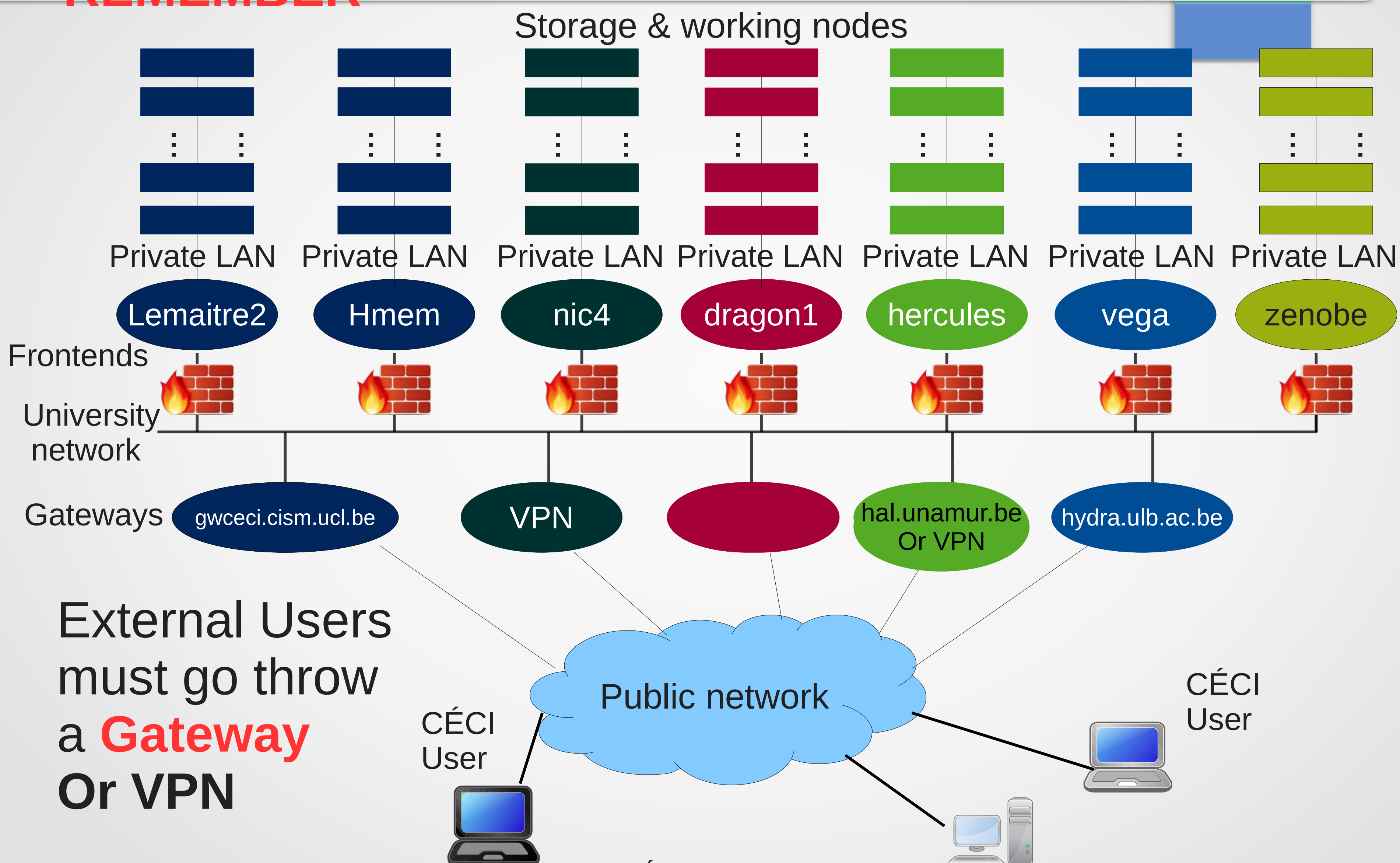
# Exercise

- Launch the ssh-agent
- Add your private key and connect.

You will be asked for you passphrase for the last time

# SSH context: CÉCI infrastructure

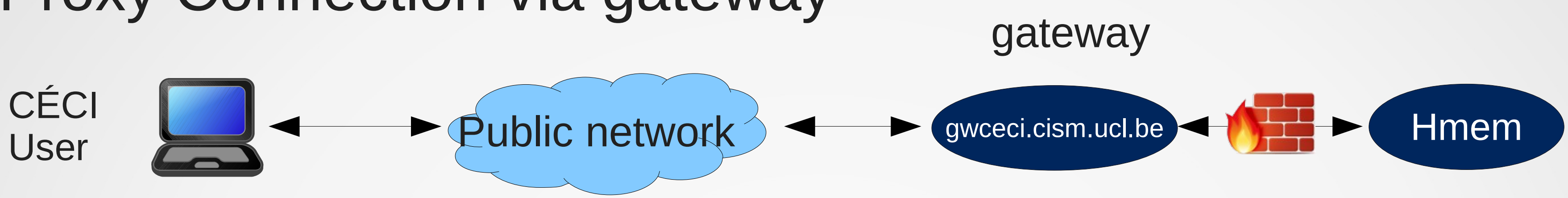
**REMEMBER**




External Users must go throw a **Gateway** **Or VPN**

# **Proxies** and (pseudo-)VPNs

## Proxy Connection via gateway




All input and output data from client is forwarded to the host through the gateway

```
$ ssh -o 'ProxyCommand ssh gatewayuser@gatewayadress -W %h:%p' hmem
```

Replace gatewayuser@gatewayadress by
your university login name and gateway address

```
$ ssh -o 'ProxyCommand ssh jcabrera@gwceci.cism.ucl.ac.be -W %h:%p' hmem
cabrera@hall.cism.ucl.ac.be's password:
Last login: Mon Aug 17 14:36:50 2015 from vm1.cism.ucl.ac.be
Welcome to
          __  __     __    __     ______     __    __
         /\ \_\ \   /\ "-./  \   /\  ___\   /\ "-./  \
         \ \  __ \  \ \ \-./\ \  \ \  __\   \ \ \-./\ \
          \ \_\ \_\  \ \_\ \ \_\  \ \_____\  \ \_\ \ \_\
           \/_/\/_/   \/_/  \/_/   \/_____/   \/_/  \/_/
                   HighMemory CISM-CECI cluster
```

# **Proxies** and (pseudo-)VPNs

## Proxy Connection via gateway

```
Gatewayadress:
- UCL: gwceci.cism.ucl.ac.be
- Unamur: hal.unamur.be
```

For UCL and UNamur user can connect through a gateway
Use the wizard http://www.ceci-hpc.be/sshconfig.html

```
# UNamur Specific -------------------------------------------------------
Host gwhal
    Hostname hal.unamur.be
    User jbcabrer
Host *%gwhal
    ProxyCommand ssh -W %h:%p gwhal
```

To connect just type:

```
$ ssh hmem%gwhal
```

You can do the same for others cluster

# Proxies and **(pseudo-)VPNs**

You can redirect throw ssh tunnel all ports for all or some of your IP connections via the gateway.
This can be done with the python program sshuttle.
To use it, **you need to have root or sudo permission**.

```
$ wget https://github.com/sshuttle/sshuttle/archive/v0.78.4.zip
$ unzip v0.78.4.zip
$ cd sshuttle-0.78.4/
$ sudo ./setup.py install
```

Redirect connections for all IP

```
$ ./sshuttle -r jbcabrer@hal.unamur.be 0.0.0.0/0
```

Now you can access to https://login.ceci-hpc.be/ from outside the university
Check IP at https://www.whatismyip.com/

Redirect only UCL IP

```
$ ./sshuttle -r gwceci 130.104.1.0/24
```

You can also install sshuttle with **pip, apt-get, yum or brew**

# SSH-based file transfer (**SCP**, rsync, SSHFS)

You can copy files/directories back and forth between computers

- Verify your agent is running and hmem is defined in your config file
- Create a temporary directory with dummy files

```
$ mkdir -p coursssh/scptest; touch coursssh/scptest/file{1..4}.txt
```

- Copy the directory to your home directory in hmem and check

```
$ scp -r coursssh/scptest hmem:coursssh/.
$ ssh hmem 'ls coursssh/scptest/'
```

- Copy it back

```
$ scp -r hmem:coursssh/scptest coursssh/scptest2
```

- Copy via proxy (from outside the universities network)

```
$ scp -r coursssh/scptest2 hmem%:coursssh/.
```

- Copy between frontends. (direct connection between frontends)

```
$ scp -r hmem:coursssh/scptest hercules:coursssh/.
```

- To use the alias hercules your ~/.ssh/config file must be set in hmem

  For a copy throw your computer use -3

```
$ scp -r -3 hmem:coursssh/scptest hercules:coursssh/.
```

# SSH-based file transfer (SCP, **rsync**, SSHFS)

rsync is widely used for backups and mirroring and as an improved copy command for everyday use

Most common usage is to synchronize files with archive option -a and compress option z. If you want to get a copy of your hard work you did in the frontend to your laptop:

```
$ ssh hmem 'mkdir coursssh/rsynctest; touch coursssh/rsynctest/file{1..4}.txt'
$ rsync -avz --progress hmem:coursssh/rsynctest coursssh/.
```

Modify a file at the frontend and synchronize

```
$ ssh hmem 'echo "Adding hello1 word in hmem" >> coursssh/rsynctest/file4.txt'
$ rsync -avz --progress hmem:coursssh/rsynctest coursssh/.
```

Modify a file in your computer and prevent Overwrite when synchronize -u

```
$ echo 'Adding hello in client' > coursssh/rsynctest/file3.txt
$ rsync -avzu --progress hmem:coursssh/rsynctest coursssh/.
```

Delete a file at the frontend and force delete it in your computer.

```
$ ssh hmem rm coursssh/rsynctest/file1.txt
$ rsync -avz --del --progress hmem:coursssh/rsynctest coursssh/.
```

# SSH-based file transfer (SCP, rsync, **SSHFS**)

Use SSHFS to mount a remote file system - accessible via SSH

## Linux install:

### Debian, Ubuntu

```
$ sudo apt-get install sshfs
```

### Fedora/CentOs

```
$ yum install sshfs
```

## Mac Install:

Install FUSE and SSHFS from https://osxfuse.github.io/

# SSH-based file transfer (SCP, rsync, **SSHFS**)

Example: Mount your CECIHOME

Create a local repository to mout the CÉCI home

```
$ mkdir CECIHOME
```

Mount the remote CÉCI Home

```
$ cluster=hmem;
$ sshfs -o uid=`id -u` -o gid=`id -g` $cluster:$(ssh $cluster 'echo $CECIHOME')/ CECIHOME
```

the command:

```
$ ssh $cluster 'echo $CECIHOME'
```

gives the path of CECIHOME in the cluster
Create file in the mounted directory

```
$ echo 'file content' > CECIHOME/file_fuse.txt
```

Check the file content in the frontend

```
$ ssh hmem 'cat $CECIHOME/file_fuse.txt'
```

disconnect

```
$ fusermount -u ~/clusters_dirs/hmem
```

# Exercise

- Mount CECIHOME from your university frontend.

# Thanks

Thank you for your attention

RFC 3447 Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1

8.2.1 Signature generation operation

RSASSA-PKCS1-V1_5-SIGN (K, M)

Input:
K signer's RSA private key
M message to be signed, an octet string

Output:
S signature, an octet string of length k, where k is the length in octets of the RSA modulus n

Errors: "message too long"; "RSA modulus too short"

Steps:

**1**. EMSA-PKCS1-v1_5 encoding: Apply the EMSA-PKCS1-v1_5 encoding operation (Section 9.2) to the message M to produce an encoded message EM of length k octets:

EM = EMSA-PKCS1-V1_5-ENCODE (M, k).

If the encoding operation outputs "message too long," output "message too long" and stop. If the encoding operation outputs "intended encoded message length too short," output "RSA modulus too short" and stop.

**2.** RSA signature:

a. Convert the encoded message EM to an integer message representative m (see Section 4.2):

m = OS2IP (EM).

b. Apply the RSASP1 signature primitive (Section 5.2.1) to the RSA private key K and the message representative m to produce an integer signature representative s:

s = RSASP1 (K, m).

c. Convert the signature representative s to a signature S of length k octets (see Section 4.1):

S = I2OSP (s, k).

3. Output the signature S.

8.2.2 Signature verification operation

RSASSA-PKCS1-V1_5-VERIFY ((n, e), M, S)

Input:
(n, e) signer's RSA public key
M message whose signature is to be verified, an octet string
S signature to be verified, an octet string of length k, where
k is the length in octets of the RSA modulus n

Output:
"valid signature" or "invalid signature"

Errors: "message too long"; "RSA modulus too short"

Steps:

1. Length checking: If the length of the signature S is not k octets, output "invalid signature" and stop.

2. RSA verification:

a. Convert the signature S to an integer signature representative s (see Section 4.2):

s = OS2IP (S).

b. Apply the RSAVP1 verification primitive (Section 5.2.2) to the RSA public key (n, e) and the signature representative s to produce an integer message representative m:

m = RSAVP1 ((n, e), s).

If RSAVP1 outputs "signature representative out of range," output "invalid signature" and stop.

c. Convert the message representative m to an encoded message EM

of length k octets (see Section 4.1):

EM' = I2OSP (m, k).

If I2OSP outputs "integer too large," output "invalid signature" and stop.

3. EMSA-PKCS1-v1_5 encoding: Apply the EMSA-PKCS1-v1_5 encoding
operation (Section 9.2) to the message M to produce a second encoded message EM' of length k octets:

EM' = EMSA-PKCS1-V1_5-ENCODE (M, k).

If the encoding operation outputs "message too long," output "message too long" and stop. If the encoding operation outputs "intended encoded message length too short," output "RSA modulus too short" and stop.

4. Compare the encoded message EM and the second encoded message EM'.
If they are the same, output "valid signature"; otherwise, output "invalid signature."

## Diffie-Hellman Key Exchange (RFC 4253)

The Diffie-Hellman (DH) key exchange provides a shared secret that cannot be determined by either party alone. The key exchange is combined with a signature with the host key to provide host authentication. This key exchange method provides explicit server authentication as defined in Section 7.

The following steps are used to exchange a key. In this, C is the client; S is the server; p is a large safe prime; g is a generator for a subgroup of GF(p); q is the order of the subgroup; V_S is S's identification string; V_C is C's identification string; K_S is S's public host key; I_C is C's SSH_MSG_KEXINIT message and I_S is S's SSH_MSG_KEXINIT message that have been exchanged before this part begins.

1. C generates a random number x ($1 < x < q$) and computes $e = g^x \bmod p$. C sends e to S.

2. S generates a random number y ($0 < y < q$) and computes $f = g^y \bmod p$. S receives e. It computes $K = e^y \bmod p$, H = hash(V_C || V_S || I_C || I_S || K_S || e || f || K) (these elements are encoded according to their types; see below), and signature s on H with its private host key. S sends (K_S || f || s) to C. The signing operation may involve a second hashing operation.

3. C verifies that K_S really is the host key for S (e.g., using certificates or a local database). C is also allowed to accept the key without verification; however, doing so will render the protocol insecure against active attacks (but may be desirable for practical reasons in the short term in many environments). C then computes $K = f^x \bmod p$, H = hash(V_C || V_S || I_C || I_S || K_S || e || f || K), and verifies the signature s on H.

Values of 'e' or 'f' that are not in the range [1, p-1] MUST NOT be sent or accepted by either side. If this condition is violated, the key exchange fails.

This is implemented with the following messages. The hash algorithm for computing the exchange hash is defined by the method name, and is called HASH. The public key algorithm for signing is negotiated with the SSH_MSG_KEXINIT messages.

First, the client sends the following:

```
byte      SSH_MSG_KEXDH_INIT
mpint     e
```

The server then responds with the following:

```
byte      SSH_MSG_KEXDH_REPLY
string    server public host key and certificates (K_S)
mpint     f
string    signature of H
```

The hash H is computed as the HASH hash of the concatenation of the following:

```
string    V_C, the client's identification string (CR and LF
          excluded)
string    V_S, the server's identification string (CR and LF
          excluded)
string    I_C, the payload of the client's SSH_MSG_KEXINIT
string    I_S, the payload of the server's SSH_MSG_KEXINIT
string    K_S, the host key
mpint     e, exchange value sent by the client
mpint     f, exchange value sent by the server
mpint     K, the shared secret
```

This value is called the exchange hash, and it is used to authenticate the key exchange. The exchange hash SHOULD be kept secret.

The signature algorithm MUST be applied over H, not the original data. Most signature algorithms include hashing and additional padding (e.g., "ssh-dss" specifies SHA-1 hashing). In that case, the data is first hashed with HASH to compute H, and H is then hashed with SHA-1 as part of the signing operation.

7. Public Key Authentication Method: "publickey" RFC 4252

The only REQUIRED authentication 'method name' is "publickey" authentication. All implementations MUST support this method; however, not all users need to have public keys, and most local policies are not likely to require public key authentication for all users in the near future.

With this method, the possession of a private key serves as authentication. This method works by sending a signature created with a private key of the user. The server MUST check that the key is a valid authenticator for the user, and MUST check that the signature is valid. If both hold, the authentication request MUST be accepted; otherwise, it MUST be rejected. Note that the server MAY require additional authentications after successful authentication.

Private keys are often stored in an encrypted form at the client host, and the user must supply a passphrase before the signature can be generated. Even if they are not, the signing operation involves some expensive computation. To avoid unnecessary processing and user interaction, the following message is provided for querying whether authentication using the "publickey" method would be acceptable.

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name in ISO-10646 UTF-8 encoding [RFC3629]
string    service name in US-ASCII
string    "publickey"
boolean   FALSE
string    public key algorithm name
string    public key blob
```

Public key algorithms are defined in the transport layer specification [SSH-TRANS]. The 'public key blob' may contain certificates.

Any public key algorithm may be offered for use in authentication. In particular, the list is not constrained by what was negotiated during key exchange. If the server does not support some algorithm, it MUST simply reject the request.

The server MUST respond to this message with either SSH_MSG_USERAUTH_FAILURE or with the following:

```
byte      SSH_MSG_USERAUTH_PK_OK
string    public key algorithm name from the request
string    public key blob from the request
```

To perform actual authentication, the client MAY then send a signature generated using the private key. The client MAY send the signature directly without first verifying whether the key is acceptable. The signature is sent using the following packet:

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name
string    "publickey"
boolean   TRUE
string    public key algorithm name
string    public key to be used for authentication
string    signature
```

The value of 'signature' is a signature by the corresponding private key over the following data, in the following order:

```
string    session identifier
byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name
string    "publickey"
boolean   TRUE
string    public key algorithm name
string    public key to be used for authentication
```

When the server receives this message, it MUST check whether the supplied key is acceptable for authentication, and if so, it MUST check whether the signature is correct.

If both checks succeed, this method is successful. Note that the server may require additional authentications. The server MUST respond with SSH_MSG_USERAUTH_SUCCESS (if no more authentications are needed), or SSH_MSG_USERAUTH_FAILURE (if the request failed, or more authentications are needed).

The following method-specific message numbers are used by the "publickey" authentication method.

```
SSH_MSG_USERAUTH_PK_OK            60
```