Universal Credit System

technical documentation

Welcome!

In this document you will find detailed technical documentation of **the Universal Credit System**. It contains all the information you need to be able to implement and/or adapt the program to your needs.

Contents

1. Dependencies

The program depends on other programs that must be installed. During setup the **`install.sh`** script will perform a check if any program is missing. The following programs are used:

name	**purpose**
awk	used to sort/filter data
basename	used to strip directory and suffix from filenames
bc	used for floating point calculations
cat	used to concatenate content
chmod	used to change file/directory permissions
cp	used to copy files
curl	used to send query to TSA and request response
cut	used to extract data from streams
date	used for date operations
dd	used to convert files
dialog	used as GUI
dirname	used to strip non-directory suffix from file name
echo	used to write output
expr	used for calculations
find	used to search files/directories
flock	used to manage read locks for multi user setups
gpg	used for transaction signing
grep	used to search files
head	used to display heading lines of a file
ls	used to list files and directories
mkdir	used to create folders and subfolders
mv	used to move files
netcat	used to send/request files
openssl	used for TSA stamp verification
printf	used to write output
rm	used to delete files
sed	used to read/modify text
sha224sum	used to hash files
sha256sum	used to hash files
sort	used to sort files
stat	used to get permissions of files/directories
tail	used to display tailing lines of a file
tar	used to create the transaction file
test	used to test files
touch	used to create files
tr	used to convert chars
umask	used to determine umask
uniq	used to filter files
wc	used to count lines, words, bytes
wget	used to fetch certificate files of TSA from Internet

2. Folders

name	description
~/assets/	**contains the assets created by the users**
~/backup/	**contains the backups created by the user**
~/certs/	**contains sub folders for TSAs with certs**
~/control/	**contains important system files**
~/keys/	**contains the public key files of the users**
~/lang/	**contains the language files**
~/proofs/	**contains subfolders for users' proofs**
~/theme/	**contains the themes for GUI**
~/trx/	**contains the trx of all users**
~/userdata/	**contains temporary user related files**

3. Details

~/assets/

Contains the assets that have been created by the users. Assets can be considered as tokens. Asset files are named like this:

<ASSET_SYMBOL>.<STAMP>

<*ASSET_SYMBOL*> is a string. <STAMP> is the stamp in UNIX timestamp format (unix epoch time).

~/backup/

Contains system backups that have been triggered by the user. With these backup files a user is able to restore it's data if corrupted. Backup files are named like this:

<STAMP>.bc

<*STAMP*> is the stamp in UNIX timestamp format (unix epoch time).

~/certs/

This folder contains a subfolder for each TSA. In this subfolder are the initial certs for the TSAs. Currently only freeTSA is supported.

Files are: cacert.pem (Certificate of CA)
tsa.pem (Certificate of TSA)

These certificates are used to verify the TSA files lying in **/proofs/**-directory via **openssl ts**.

~/control/

This folder contains important system files that are used by UCS. Files in this folder:

config.conf	configuration file
uca.conf	UCA list
tsa.conf	TSA list
install_config.conf	default configuration file
dh.db	Database for Perfect Forward Secrecy
install.dep	installation dependencies
keyring.file	GPG Keyring
HELP.txt	help text

Furthermore all private keys of accounts that have been created on this machine are saved under ***~/control/keys/*** folder.

~/keys/

This folder contains the public keys for all users. The keys are 4096bit RSA keys. Creation of these keys is done via GnuPG while all keys are managed in ***keyring.file*** in scripts's ***~/control/*** directory. Keys are named as follows:

<ADDRESS>.<STAMP>

<ADDRESS> is actually a SHA-256 hash that is calculated by hashing the string „*<ACCOUNTNAME>_<PIN>_<STAMP>*“ while *<STAMP>* is the point of creation in UNIX timestamp format (unix epoch time). The calculation of the hash is also used during logon. The account name and PIN are used along with the stamps to find a matching key.

~/lang/

New language files will be imported automatically and can be selected in GUI. They will be sourced within the script at the start.
Naming of the lang-files is:

lang_<language-short>_<language-long>.conf

<language-short> is the country code e.g. ***EN*** for ***ENGLISH*** or **FR** for ***FRANCE*** while *<language-long>* is the name of the language in the foreign language e.g. ***SVENSKA*** for *SWEDISH* or ***ITALIANO*** for *ITALIAN*. You can add new files to this folder and the script will automatically include them in the list of languages available.

Make sure you don’t change/remove the tags '<tag>' or '/n'! These tags are being used to format the graphical user interface provided by `dialog`!

Everything else can be adopted to the related language. If new language files are named like described above the script is able to recognize them and you are able to select them in GUI.

~/proofs/

This folder contains a subfolder for each user and in this subfolder are the users' proofs (index-file, TSA query, TSA response). The subfolders are in following format (per user):

/<ADDRESS>.<STAMP>/

<ADDRESS>.<STAMP> is equal to the user. The verification of the users TSA files in this folder is done via the **openssl ts** command:

freetsa.tsq	(freeTSA query)
freetsa.tsr	(freeTSA response)
<ADDRESS>.<STAMP>.txt	(index file)

The index file contains a list of all acknowledged users with their proofs and trx. The index file if verified using GnuPG.

~/theme/

This folder contains the themes used by the dialog command. Any theme placed in this folder will be automatically recognized and is available in main menu → settings → themes.

~/trx/

This folder contains the transactions of all users. transaction naming:

<ADDRESS>.<USER_STAMP>.<TRX_STAMP>

<ADDRESS>.<USER_STAMP> is the address of the sender while *<TRX_STAMP>* is the creation date of the transaction in UNIX timestamp format (unix epoch time). Transactions are simple text files containing a header that includes all related information and a signature of the sending user.

~/userdata/

Contains a subfolder for each user that logged on on this machine. In this subfolder temporary user specific files:

all_assets.dat	list of all assets
all_keys.dat	list of all keys
all_accounts.dat	list of all accounts
all_trx.dat	list of all transactions
blacklisted_accounts.dat	list of all deleted accounts
blacklisted_trx.dat	list of all deleted transactions
depend_accounts.dat	list of all depending accounts
depend_trx.dat	list of all depending transactions
depend_confirmations.dat	list of all depending transactions without enough confirmations
<YYYYMMDD>_ledger.dat	ledger file of corresponding date
<YYYYMMDD>_scoretable.dat	score file of corresponding date
<YYYYMMDD>_index_trx.dat	list of all transactions the user has acknowledged on corresponding date

Files that have been extracted from transaction files or sync files are also stored there under ***~/userdata/<USER_NAME>/temp/***. These files are being extracted into the /temp-folder. From there they are either moved into the related folders or being deleted.

4. EXAMPLE TRANSACTION

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA512

:TIME:1665066890
:AMNT:1.000000000|UCC
:SNDR:9d8c98a97b2c3e689afef90310a35130bde86fd6f43ef6764b391c40ba37f8dd.1613477808
:RCVR:ca2c6f1d030c0ea7e56893a89c32d6c86478b56ff40cfb327608ef47a58bc401.1613477644
:PRPS:86954a568d85d4e6b73569f5f8b2c44a956ba5b7acfcf55fca29159c
-----BEGIN PGP SIGNATURE-----

iQIzBAEBCgAdFiEEDWT9IMjKKYd6rKhIT+61NsV5pQUFAmM+54sACgkQT+61NsV5
pQXtXxAAwlE6Eift7OA+k+3CC0lzeBRjpqRoJmMKkeyH9dXjyHf2nEThx3mFADJ0
mppAPkrCHh9qGicsbv4xwU9ciXLClNgNu3RaZ3GrHPKfJR3FUTy/xKVRSaw/+S19
votpKy0HlLb6Lke9gLDjOhVLbQ0/EqwGfbLT2/mOiAavdKKmpDBc8QBqWyfFY8+B
pQNDXXESz68FX2hboFOvRMvyF0QRzFMNrpsjkG4W1hiSAoJ5ljKBokneDDsnV5FC
I887ds8gKu2sEpTYxwy2jI4M5oa39g6gtk/bJh/b3HxTk3QNULkIwrvOGes3KXfi
6ik8X2p37yeBD/HYNpH2c5ViHqAvjlFdwemShZWSX9UXTKGHBry2dNROx/yPzib4
rsHFKN/2l5UQ/0OwA0V8s8eEDpIYbrbQ/KqSLrXW45jGwI/w/tWJe4gqRPM27JAg
PYic+FwLUnYeEF93RIKsPB/Wtoa4vCsO8pPjvgVDav8WQh9YmyPmsU4AjMo7i4Jl
4hHg+R1ttRbXCBz9PQ+TxQHRFfOFaJRfEifHj2Aqli4PCxuWLx047UTc6PxlBE0w
cwHgz3Of0suc5DEyXhDD+eBSMUzNS4/NMlT30VvOauoMnoWDrTFgTddqlnX65T7O
V1Vkf3t5n99ys/wz9qRnRzFAHIAN0yfYfdApqO7K4+RUXktQW1s=
=g2B2
-----END PGP SIGNATURE-----
```

A transaction is actually a clearsigned text file in OpenPGP format. It contains all necessary information: the *date of creation* (TIME), the *amount to transfer* with asset definition (AMNT), the *sender of the transaction* (SNDR), the *receiver of the transaction* (RCVR) and a *purpose* (PRPS).

5. Wallet installation

Assuming you use the packaging tool APT, the command **`apt-get install`** is used. Please note that if you are using any other packaging tool than APT the command for installing a package might be different. In this chase change **`apt-get install`** to the command your packaging tool is using!

Install Git (you may use **`sudo`** in front):

```
apt-get install git
```

Create a directory wherever you want and step into this directory:

```
mkdir ucs
cd ucs
```

Clone the GitHub repository and step into this directory:

```
git clone https://github.com/universal-credit-system/wallet
cd wallet/
```

Now you can execute the **`install.sh`** script. The script will check for depending programs and if all depending programs are installed the setup will continue. If there is a program that needs to be installed the script will output the program names and then quit. In this case you have to install these programs first and then run install.sh script again:

```
./install.sh
```

After setup you can run **`ucs_client.sh`**:

```
./ucs_client.sh
```

6. Assets

The universal credit system supports the creation and use of tokens, here so called 'assets'. Assets can be fungible or non-fungible. All assets are stored under **~/assets/**. The main difference between the currency UCC and assets is that there is **no scoring** done when transferring assets.

FUNGIBLE ASSETS

Fungible assets are assets that can be converted by all users. This means, that every user can exchange his balance to this asset. The exchange of *fungible assets* is unlimited; there is no maximum amount that can be exchanged/converted. Below is an example of a fictitous fungible asset 'TestFungibleToken' having the asset-symbol 'TFT.1655676000 ':

example fungible asset `'TFT.1655676000'`:

```
asset_description=TestFungibleToken
asset_price=2.000000000
asset_fungible=1
```

asset_description is the description of the assets, *asset_price* is the price per unit in UCC. *asset_fungible=1* defines that this is a fungible asset. The import of fungible assets that other users have created is **disabled** per default. To enable the auto-import of fungible assets you have to modify the file `control/config.conf` and set `'import_fungible_assets=1'`.

NON-FUNGIBLE ASSETS

Non-fungible assets are assets that cannot be converted/exchanged. To posess such a asset you either have to receive it from a owner or you have to be the initial owner yourself. The total amount of a non-fungible asset is defined as value **asset_quantity**. Below is an example of a fictitous non-fungible asset 'TestNonFungibleToken' having the asset-symbol 'TNFT.1655676000':

example non-fungible asset`'TNFT.1655676000'`:

```
asset_description=TestNonFungibleToken
asset_owner=9d8c98a97b2c3e689afef90310a35130bde86fd6f43ef6764b391c
40ba37f8dd.1613477808
asset_quantity=100.000000000
asset_fungible=0
```

asset_description is the description of the assets, *asset_owner* is the initial owner, *asset_quantity* is the number of tokens. *asset_fungible=0* defines, that this is a non-fungible asset. The import of fungible assets that other users have created is **disabled** per default. To enable the auto-import of fungible assets you have to modify the file `control/config.conf` and and set `'import_non_fungible_assets=1'`.

7. HOW TO SET UP A DAO

This topic is about how you can set up a *decentralized autonomous organization* (DAO).

To understand how UCS has implemented the idea of DAOs simply imagine the stocks of a company: depending on the number of stocks you own you also own a part of that company. At some point the revenue of this company will be distributed among the owners of the stocks. The percentage of the revenue your will get is equal to the percentage of the company you own by the stocks. In UCS the concept of DAOs is limited to the fair distribution of amounts sent to that DAO among the DAO participants.

From a technical point of view a DAO consists of two assets: a fungible asset used to receive the revenues and a fungible or non-fungible asset that is used to reflect the owners. So the ‘stocks’ are simply fungible or non-fungible assets! Every owner of such a fungible or non-fungible asset will be considered as ‘stock owner’. The asset that is used to reflect these property rights is defined as ‘asset_owner=’ of the asset that is used to receive the revenues. The first asset acts as central receiver address for the owners of the second asset.

The DAO has no voting process about what to do with the balance like it is done within the ethereum DAO. The users have to make agreements outside UCS and then forward the balances themself to a second DAO that is handling the specific investment.

STEP 1 : SET UP A FUNGIBLE ASSET

This asset will act as address for all owners of asset DAO.1655676000.

example fungible asset '**DAOT.1655676000':**

```
asset_description=DAOToken
asset_owner=DAO.1655676000
asset_price=1.000000000
asset_fungible=1
```

STEP 2 : SET UP A FUNGIBLE OR NON FUNGIBLE ASSET

Choose a *non-fungible asset* if you wish to have a fix number of property rights that cannot be increased. This means that other users cannot join the DAO on their own.

example non-fungible asset '**DAO.1655676000**':

```
asset_description=DAO
asset_quantity=1000.000000000
asset_owner=9d8c98a97b2c3e689afef90310a35130bde86fd6f43ef6764b391c40ba37f
8dd.1613477808
asset_fungible=0
```

Choose a *fungible asset* if you want to have no fix number of property rights. This means any other user can join the DAO by transferring a amount to this asset:

example fungible asset '**DAO.1655676000**':

```
asset_description=DAO
asset_price=1.000000000
asset_fungible=1
```

8. HOW TO CMD-MODE

Below you can find detailed examples of commands for cmd-mode. With these commands it is e.g. very easy to set up automated payment solutions.

HOW TO CREATE A USER

```
EXAMPLE COMMAND:
./ucs_client.sh -action create_user -user TESTUSER -password
TESTPASSWORD

OUTPUT:
USER:<ACCOUNTNAME>
PIN:<PIN>
PASSWORD:>PW<          # NOTE: PW PUT IN ><
ADRESS:<ADRESS>
KEY:<KEYFILE>
KEY_PUB_HOME:ACCOUNTNAME_PIN_STAMP_pub.asc
KEY_PRV_HOME:ACCOUNTNAME_PIN_STAMP_priv.asc
```

NOTE: Currently the exported private is always stored in script **/control/keys/**-folder while the public key is always stored in **/keys**-folder. So **if you handover a path where these keys should be stored, it will not be used! These exported keys are your public and private backup keys - you better keep them under your pillow!** With these keys you will be able to restore your account if everything is lost.

HOW TO CREATE A SMALL TRANSACTION (only pack new files, if possible)

```
EXAMPLE COMMAND:
./ucs_client.sh -action create_trx -user TESTUSER -pin 12345
-password TESTPASSWORD -receiver ADRESS -amount 1.000000000 -asset
ASSET -purpose ‶PURPOSE TEXT‶ -type partial -path
/path/to/outputdir
```

NOTE: Type “partial” means the program will check whether sender and receiver have common transaction knowledge and if so it will only add data to the transaction file that are new to the sender. This can reduce the size of a transaction file.

HOW TO CREATE A BIG TRANSACTION (pack all files)

```
EXAMPLE COMMAND:
./ucs_client.sh -action create_trx -user TESTUSER -pin 12345
-password TESTPASSWORD -receiver ADRESS -amount 1.000000000 -asset
ASSET -purpose ‶PURPOSE TEXT‶ -type full -path /path/to/outputdir
```

NOTE: Type “full” means it will pack all data independent of common transaction knowledge.

HOW TO PARTIALLY READ A TRANSACTION FILE (only unpack new files)

```
EXAMPLE COMMAND:
./ucs_client.sh -action read_trx -user TESTUSER -pin 12345
-password TESTPASSWORD -type partial -path /path/to/file/file.trx
```

NOTE: Type “partial” means the program will check whether sender and receiver have common transaction knowledge and if so it will only unpack data that are new to the sender. <u>This is standard and you should always do it this way to avoid that other files you already have being overwritten.</u>

HOW TO FULLY READ A TRANSACTION FILE (unpack all files):

```
EXAMPLE COMMAND:
./ucs_client.sh -action read_sync -user TESTUSER -pin 12345
-password TESTPASSWORD -type full -path /path/to/file/file.trx
```

NOTE: Type “full” means the program unpacks all data of the transaction file. <u>This overrides your existing data and should only be done with a lot of precaution and awareness!</u> E.g. this allows you to restore you data by a transaction file only if corrupted. <u>BE CAREFUL WITH THIS!</u>

HOW TO CREATE A SYNCRONISATION FILE (contains all files):

```
EXAMPLE COMMAND:
./ucs_client.sh -action create_sync -user TESTUSER -pin 12345
-password TESTPASSWORD -path /path/to/outputdir
```

NOTE: As there is no explicit receiver for a synchronization file it always contains all data of all users. It is up to the receiver of the file which data to extract (full or partial).

HOW TO PARTIALLY READ A SYNCRONISATION FILE (only unpack new):

```
EXAMPLE COMMAND:
./ucs_client.sh -action read_sync -user TESTUSER -pin 12345
-password TESTPASSWORD -type partial -path /path/to/file/file.sync
```

NOTE: Type “partial” means the program will check whether sender and receiver have common transaction knowledge and if so it will only unpack data that are new to the sender. <u>This is standard and you should always do it this way to avoid that other files you already have being overwritten.</u>

HOW TO FULLY READ A SYNCRONISATION FILE (unpack all):

```
EXAMPLE COMMAND:
./ucs_client.sh -action read_sync -user TESTUSER -pin 12345
-password TESTPASSWORD -type full -path /path/to/file/file.sync
```

NOTE: Type "full" means the program unpacks all data of the synchronization file. This overrides your existing data and should only be done with a lot of precaution and awareness! E.g. this allows you to restore you data by a synchronization file only if corrupted. BE CAREFUL WITH THIS!

HOW TO SYNC WITH UCA

```
EXAMPLE COMMAND:
./ucs_client.sh -action sync_uca -user TESTUSER -pin 12345
-password TESTPASSWORD
```

NOTE: The action "sync_uca" will create no output if successful and will always exit with code 0 even if the receive/send of data to the defined UCA(s) failed. If receive/send to one more of UCAs failed it will output a "ERROR" message containing used IP (<uca_ip>) and Port (<ucs_snd_port>) as defined in ~/control/uca.conf. .

HOW TO CREATE A BACKUP:

```
EXAMPLE COMMAND:
./ucs_client.sh -action create_backup
```

HOW TO RESTORE A BACKUP:

```
EXAMPLE COMMAND:
./ucs_client.sh -action restore_backup -path
/path/to/ucs/backup/<STAMP>.bcp
```

HOW TO DISPLAY STATISTICS:

```
EXAMPLE COMMAND:
./ucs_client.sh -action show_stats -user TESTUSER -pin 12345
-password TESTPASSWORD
```

9. Universal Credit Contractor

The contractor acts as a wrapper script for the **`ucs_client.sh`** script and allows the user to set up smart contracts for transactions. When executed the bash script **`ucs_contractor.sh`** will source the logic and perform actions based on the logic and the ruleset.

WHAT IS A CONTRACT?

A contract always consists of **at least** two files:

- a file that contains the actual logic in the folder **`/contracts/`** (that contains a definition of a function called **`contract_action()`** which is sourced)

- a file that contains the ruleset in the folder **`/rulesets/`** (definitions of variables used by the logic)

Both files are handed over to **`ucs_contractor.sh`** via parameters. At execution, the logic of the contract is loaded and controlled with the variables of the rulesets file. In principle, any logic can be implemented.

HOW TO SETUP

STEP 1 : GET THE SOURCES

To run the contractor you need to have a full client set up with a user. See GitHub Readme how to install and run the client. Assuming you already have the client, step into ths directory and unpack the contractor:

```
tar -xvf contractor.tar
```

The tarball contains the following files/folders:

- the script **`ucs_contractor.sh`**
- the folder **`/contracts/`**
- the folder **`/rulesets/`**
- the file **`/control/contractor_HELP.txt`**

The tarball also contains some example contracts (cashier, filter, accountant and tombola) and related rulesets for these contracts.

STEP 2 : DEFINE YOUR CONTRACT(S)

Create a smart contract logic and a ruleset based on your needs.

EXAMPLE

The following example is a ruleset for the supplied smart contract **`accountant.logic`**. The smart contract **`accountant.logic`** acts as a simple accountant sending transactions based on received transactions. Only parameters related to the transaction can be defined as triggers and the action is limited to the creation of new transaction(s).

See below example ruleset `accountant.ruleset`:

```
ruleset_asset="YOUR_ASSET_HERE"
ruleset_sender="*"
ruleset_receiver="YOUR_ADRESS_HERE"
ruleset_amount="*"
ruleset_amount_comparison_operator=""
ruleset_amount_comparison_variable=""
ruleset_purpose="*"
ruleset_required_confirmations=0
contract_asset="${trx_asset}"
contract_sender="YOUR_ADRESS_HERE"
contract_sender_password="YOUR_PASSWORD_HERE"
contract_receiver="${trx_sender}"
contract_amount="${trx_amount}"
contract_purpose=`echo "${trx_file}"|sha256sum|cut -d ' ' -f1`
contract_type="partial"
alias send_trx='${script_path}/ucs_client.sh -action create_trx
-sender ${contract_sender} -password "${contract_sender_password}"
-receiver ${receiver} -amount ${contract_amount} -asset $
{contract_asset} -purpose ${contract_purpose} -type $
{contract_type}'
```

The above ruleset ensures that the smart contract `accountant.logic` sends all transactions that were sent to you back to the sender (if you enter a asset, your address and your password).

`accountant.logic` will look for transactions:

- matching the defined asset (`ruleset_asset="YOUR_ASSET_HERE"`)
- having any sender (`ruleset_sender="*"`)
- you as receiver (`ruleset_receiver="PUT_YOUR_ADRESS_HERE"`)
- any amount (`ruleset_amount="*"`)
- any purpose (`ruleset_purpose="*"`)
- with no confirmations (`ruleset_required_confirmations=0`)

If one or multiple transactions match this criteria the contractor will create a transaction:

- having the initial asset that was sent as asset to send (`contract_asset="${trx_asset}"`)
- having the initial amount that was sent as amount to send (`contract_amount="${trx_amount}"`)
- the initial sender as receiver (`contract_receiver="${trx_sender}"`)
- with a sha256 hash of the initial trx filename as purpose (`` contract_purpose=`echo "${trx_file}"|sha256sum|cut -d ' ' -f1 ``)
- the transaction type is ‘partial’.

And finally the alias “**send_trx**” is defined. That is actually the action that is being triggered.

STEP 3 : SCHEDULE THE CONTRACTOR

You either manually execute the **ucs_contractor.sh** or schedule a job for this e.g. with CRON. To execute your contract simply handover your ruleset and your contract with full path:

```
./ucs_contractor.sh -ruleset /path/to/contract.ruleset
-contract /path/to/contract.logic
```

Please note that **accountant.logic** will create no output If no transaction matched the ruleset.

To display the help text run:

```
./ucs_contractor.sh -help
```

More information:

At execution the **ucs_contractor.sh** script will check if a contract file is there (parameter **-contract <PATH>**) and if a ruleset file is there (parameter **-ruleset <PATH>**). If both files are there the function **contract_action()** of the contract logic file will be sourced and called. That’s all.

This means that if you need a ruleset file the logic to source/read it must be placed within the contract logic (see **accountant.logic** and **tombola.logic**). So the ruleset file is **NOT** sourced within **ucs_contractor.sh**. The contractor only loads the logic file and calls the function within that file. This logic file can contain whatever you want. All triggers and actions must be defined in a function named **contract_action()**.

Depending on what you want to do you might not need a ruleset file, but the ucs_contractor.sh will still check if that file is there. A solution would be to handover the same path for **-ruleset** as for **-contract**.

The fact that contract logic and ruleset are separate files allows the user to run the same contract logic with different rulesets!

10. UCA Link Servers

UCS allows user to set up their own UCA link servers. UCA link servers can be considered as nodes that help spreading transaction knowledge by automating the sync process.

STEP 1 : GET THE SOURCES

First of all you have to get D. J. Bernstein's **ucspi-tcpserver** (see http://cr.yp.to/ucspi-tcp.html). There are several ways to get it run. While you can `make` your own Build we have installed the **ucspi-tcp-ipv6** debian package via `apt-get`.

After you have installed the client, step into this directory and extract the server files:

```
tar -xvf server.tar
```

The following files will be extracted:

```
control/server.conf
controller.sh
logwatch.sh
filewatch.sh
start_server.sh
stop_server.sh
sender.sh
receiver.sh
```

Also the folders `/log/` and `/server/` will be extracted. The folder `/log/` is where the server will write the logfiles to. In the folder `/server/` the temporary files of the server are stored.

STEP 2 : CUSTOMIZE THE SERVER

Modfiy the file `control/server.conf` . At least you have to enter your IP-address and the logon credentials (username, PIN, password) of the user that should be used by the server.

STEP 3 : PUBLISH YOUR UCA LINK SERVER DETAILS

Add a line to `uca.conf` file in the `/control/` folder.
The format should be:

```
IP_OR_URL,SEND_PORT,RECEIVE_PORT,DESCRIPTION,
```

like e.g.:

```
127.0.0.1,15000,15001,CUSTOM SERVER,
```

Send this to your friends and the people that you want to use your server. You could also publish the details in this Forum or somewhere else on the web.

STEP 4 : START THE SERVER

You start the server by running the **`start_server.sh`** script:

```
./start_server.sh
```

The script will start **`controller.sh`** that acts as daemon running and monitoring the scripts **`sender.sh`**, **`receiver.sh, logwatch.sh`** and **`filewatch.sh`** in the background. People can now automatically sync with you by using the UCA link functionality.

STEP 5 : STOP THE SERVER

You stop the server by running the **`stop_server.sh`** script:

```
./stop_server.sh
```

11. Universal Credit Webwallet

The webwallet is an easy built solution to provide access to the wallet via a webpage. It was developed and tested on a setup with NGINX and PHP-FPM (FastCGI).

The users start at a landing page named **`index.html`**. The users credentials are send via POST method to the script **`wallet.php`** which runs on the server side. The script itself then calls a shell script named **`webwallet.sh`**. The **`webwallet.sh`** script acts as a connector between the wallet client and the webserver. The webwallet script triggers the calls of **`ucs_client.sh`**, catches the output and builds a webpage for the user based on that data. The script basically outputs html code to STDOUT that is then forwarded to the webserver.

WEBWALLET INSTALLATION

STEP 1 : HAVE A RUNNING NGINX WITH PHP-FPM SETUP

Once you have a working set up it is important to increase the *timeouts* that are set in the NGINX config. Add the following lines to your NGINX server config.:

```
proxy_read_timeout 300;
proxy_connect_timeout 300;
proxy_send_timeout 300;
```

In the PHP section of the NGINX server config you have to add the following line right below the line containing **`fastcgi_pass`**:

```
fastcgi_read_timeout 300s;
```

These timeout values can be different, but keep in mind that depending on your hardware it could take some minutes for the script to calculate everything. So to avoid getting a server timeout we suggest to set this value to few minutes. You also have to make sure that the user under which PHP is running has writeaccess to the wallets home directory because files are uploaded into this folder.

STEP 2 : EXTRACT THE SOURCES

Step into the wallet home directory and unpack **`webwallet_home.tar`**:

```
tar -xvf webwallet_home.tar
```

After that extract the file **`webwallet_www-data.tar`**. The target folder that is used in below command is your webservers directory (**`/var/www/html`**). If your webserver uses a different directory you have to change the path after ‘-C’ option to the one that your webserver is using (make sure you have write permissions for this directory ! If you don’t have these permissions use **sudo** in front of this command):

```
tar -xvf webwallet_www-data.tar -C /var/www/html
```

STEP 3 : RUN THE INSTALL SCRIPT

Now run the installer script (the user that runs this scirpt must have write access to the webservers directory e.g. /var/www/html so you may used **`sudo`** in front and again change /var/www/html to the directory your webserver is using if it differs):

```
sudo ./install_webwallet.sh /var/www/html
```

STEP 4 : START NGINX AND PHP-FPM

Start NGINX and PHP-FPM and you should be able to access the webwallet via browser.