Contents

NAME

tsshbatch - Run Commands On Batches Of Machines

Warning

`tsshbatch` is a powerful tool for automating activities on many servers at a time. This also gives you to power *to make many mistakes at a time!* This is especially true if you have `sudo` privilege promotion capabilities on the systems in your care. *So be careful out there!*
We therefore STRONGLY recommend you do the following things to mitigate this risk:

- Read This Fine Manual from beginning to end.
- Practice using `tsshbatch` on test machines or VMs that can easily be recovered or reimaged if you break someting.
- Make heavy use of test mode (which is the default) to see what the program *would* do if it actually ran in execution mode.

SYNOPSIS

```
tsshbatch.py [-EKNSTaehkqstvxy -G 'file dest' -P 'file dest' -f cmdfile -l logfile -n name -p pw
```

DESCRIPTION

`tsshbatch` is a tool to enable you to issue a command to many hosts without having to log into each one separately. When writing scripts, this overcomes the `ssh` limitation of not being able to specify the password on the command line.

You can also use `tsshbatch` to `GET` and `PUT` files from- and to many hosts at once.

`tsshbatch` also understands basic `sudo` syntax and can be used to access a host, `sudo` a command, and then exit.

`tsshbatch` thus allows you to write complex, hands-off scripts that issue commands to many hosts without the tedium of manual login and `sudo` promotion. System administrators, especially, will find this helpful when working in large server farms.

OPTIONS

`tsshbatch` supports a variety of options which can be specified on either the command line or in the `$TSSHBATCH` environment variable:

-B — Print start, stop, and elapsed execution time statistics. This does not include any time spent for interactive prompting and response, but reflects actual program runtime. (Default: Off)

-C configfile — Specify the location of the ssh configuration file. (Default: `~/.ssh/config`)

-E — Normally, `tsshbatch` writes it's own errors to `stderr`. It also writes the `stderr` output from each host it contacts to the local shell's `stderr` (unless the `-e` option has been selected).

The `-E` option redirects any such `tsshbatch` output intended for `stderr` to `stdout` instead. This avoids the need to do things like `2>&1 | ...‘ on the command line when you want to pipe all ‘‘tsshbatch` output to another program.

-K Force prompting for passwords. This is used to override a prior `-k` argument.

-G spec GET file on host and write local dest directory. `spec` is a quoted pair of strings. The first specifies the path of the source file (on the remote machine) to copy. The second, specifies the destination *directory* (on the local machine):

```
tsshbatch.py -G "/foo/bar/baz /tmp" hostlist
```

This copies `/foo/bar/baz` from every machine in `hostlistfile` to the local `/tmp/` directory. Since all the files have the same name, they would overwrite each other if copied into the same directory. So, `tsshbatch` prepends the string `hostname-` to the name of each file it saves locally.

-H hostlistfile List of hosts on which to run the command. This should be enclosed in *quotes* so that the list of hosts is handed to the -H option as a single argument:

```
-H 'host1 host2 host3'
```

-N Force interactive username dialog. This cancels any previous request for key exchange authentication.

-P spec PUT file from local machine to remote machine destination directory. `spec` is a quoted pair of strings. The first specifies the path of the source file (on the local machine) to copy. The second, specifies the destination *directory* (on the remote machine):

```
tsshbatch.py -P "/foo/bar/baz /tmp" hostlist
```

This copies `/foo/bar/baz` on the local machine to `/tmp/` on every host in `hostlist`.

-S Force prompting for `sudo` password.

-T seconds Set timeout for ssh connection attempts. (Default: 15 seconds)

-a Don't abort program after failed file transfers. Continue to next transfer attempt. (Default: Abort)

-b Don't abort program after failed `sudo` command. Normally, any `sudo` failure causes immediate program termination. This switch tells `tsshbatch` to continue processing on the next host even if such a failure occurs. This allows processing to continue for those hosts where `sudo` does work correctly. This is helpful in large environments where `sudo` is either improperly configured on some hosts or has a different password. This can also be used to discover where `sudo` does- and does not work correctly.

-e Don't report remote host `stderr` output.

-f cmdfile Read commands from a file. This file can be commented freely with the `#` character. Leading- and trailing whitespace on a line are ignored.

-h Print help information.

-k Use ssh keys instead of name/password credentials.

-l logfile Log diagnostic output to `logfile`. (Default: `/dev/null`)

-n name	Login name to use.
-p pw	Password to use when logging in and/or doing `sudo`.
-q	Quiet mode - produce less noisy output. Turns off `-y`.
-s	Silence all program noise - only return command output. Applies only to command operations. File transfer and error reporting, generally, are unaffected.
-t	Test mode: Only show what *would* be done but don't actually do it. This also prints diagnostic information about any variable definitions, the list of hosts, any `GET` and `PUT` requests, and final command strings after all variable substitutions have been applied. This is the default program behavior.
-v	Print detailed program version information and exit.
-x	Override any previous `-t` specifications and actually execute the commands. This is useful if you want to put `-t` in the `$TSSHBATCH` environment variable so that the default is always run the program in test mode. Then, when you're ready to actually run commands, you can override it with `-x` on the command line.
-y	Turn on 'noisy' reporting for additional detail on every line, instead of just at the top of the `stdout` and `stderr` reporting. This is helpful when you are filtering the output through something like `grep` that only returns matching lines and thus no context information. Turns off `-q`.

If the `-H` option is not selected, the item immediately following the options is understood to be the name of the `hostlistfile`. This is a file that contains the name of each host - one per line - on which to run the commands. This file can be commented freely with the `#` character. Leading- and trailing whitespace on a line are ignored.

The last entry on the command line is optional and defines a command to run. `tsshbatch` will attempt to execute it on every host you've specified either via `-H` or a `hostlistfile`:

```
tsshbatch.py -Hmyhost ls -al /etc
```

This will do a `ls -al /etc` on `myhost`.

Be careful when using metacharacters like `&&, <<, >>, <, >` and so on in your commands. You have to escape and quote them properly or your local shell will interfere with them being properly conveyed to the remote machine.

If you've specified a `cmdfile` containing the commands you want run via the `-f` option, these commands will run *before* the command you've defined on the command line. It is always the last command run on each host.

You can put as many `-f` arguments as you wish on the command line and the contents of these files will be run in the order they appeared from left-to-right on the command line.

`tsshbatch` does all the `GETs`, then all the `PUTs` before attempting to do any command processing. If no `GETs`, `PUTs`, or commands have been specified, `tsshbatch` will exit silently, since "nothing to do" really isn't an error.

ENVIRONMENT

`tsshbatch` respects the `$TSSHBATCH` environment variable. You may set this variable with any options above you commonly use to avoid having to key them in each time you run the program. For example:

```
export TSSHBATCH="-n jluser -p l00n3y"
```

This would cause all subsequent invocations of `tsshbatch` to attempt to use the login name/password credentials of `jluser` and `l00n3y` respectively.

`tsshbatch` also supports searching for files over specified paths with the `$TSSHBATCHCMDS` and `$TSSHBATCHHOSTS` environment variables. Their use is described later in this document.

SSH CONFIGURATION FILE PROCESSING

`tsshbatch` has limited support for ssh configuration files. Only the `HostName` and `IdentityFile` directives are currently supported.

By default, `tsshbatch` will look in `~/.ssh/config` for this configuration file. However, the location of the file can be overriden with the `-C` option.

FEATURES AND USE CASES

The sections below describe the various features of `tsshbatch` in more detail as well as common use scenarios.

Different Ways To Specify Targeted Hostnames

There are two ways to specify the list of hosts on which you want to run the specified command:

- On the command line via the `-H` option:

  ```
  tsshbatch.py -H 'hostA hostB' uname -a
  ```

  This would run the command `uname -a` on the hosts `hostA` and `hostB` respectively.

  Notice that the list of hosts must be separated by spaces but passed as a *single argument*. Hence we enclose them in single quotes.

- Via a host list file:

  ```
  tsshbatch.py myhosts df -Ph
  ```

  Here, `tsshbatch` expects the file `myhosts` to contain a list of hosts, one per line, on which to run the command `df -Ph`. As an example, if you want to target the hosts `larry`, `curly` and `moe` in `foo.com`, `myhosts` would look like this:

  ```
  larry.foo.com
  curly.foo.com
  moe.foo.com
  ```

  This method is handy when there are standard "sets" of hosts on which you regularly work. For instance, you may wish to keep a host file list for each of your production hosts, each of your test hosts, each of your AIX hosts, and so on.

  You may use the `#` comment character freely throughout a host list file to add comments or temporarily comment out a particular host line.

You can even use the comment character to temporarily comment out one or most hosts in the list given to the `-H` command line argument. For example:

```
tsshbatch.py -H "foo #bar baz" ls
```

This would run the `ls` command on hosts `foo` and `baz` but not `bar`. This is handy if you want to use your shell's command line recall to save typing but only want to repeat the command for some of the hosts your originally Specified.

Authentication Using Name And Password

The simplest way to use `tsshbatch` is to just name the hosts can command you want to run:

```
tsshbatch.py linux-prod-hosts uptime
```

By default, `tsshbatch` uses your login name found in the `$USER` environment variable when logging into other systems. In this example, you'll be prompted only for your password which `tsshbatch` will then use to log into each of the machines named in `linux-prod-hosts`. (*Notice that his assumes your name and password are the same on each host!*)

Typing in your login credentials all the time can get tedious after awhile so `tsshbatch` provides a means of providing them on the command line:

```
tsshbatch.py -n joe.luser -p my_weak_pw linux-prod-hosts uptime
```

This allows you to use `tsshbatch` inside scripts for hands-free operation.

If your login name is the same on all hosts, you can simplify this further by defining it in the environment variable:

```
export TSSHBATCH="-n joe.luser"
```

Any subsequent invocation of `tsshbatch` will only require a password to run.

HOWEVER, there is a huge downside to this - your plain text password is exposed in your scripts, on the command line, and possibly your command history. This is a pretty big security hole, especially if you're an administrator with extensive privileges. (This is why the `ssh` program does not support such an option.) For this reason, it is strongly recommended that you use the `-p` option sparingly, or not at all. A better way is to push ssh keys to every machine and use key exchange authentication as described below.

However, there are times when you do have use an explicit password, such as when doing `sudo` invocations. It would be really nice to use `-p` and avoid having to constantly type in the password. There are two strategies for doing this more securely than just entering it in plain text on the command line:

- Temporarily store it in the environment variable:

  ```
  export TSSHBATCH="-n joe.luser -p my_weak_pw"
  ```

  Do this *interactively* after you log in, not from a script (otherwise you'd just be storing the plain text password in a different script). The environment variable will persist as long as you're logged in and disappear when you log out.

  If you use this just make sure to observe three security precautions:

  1) Clear your screen immediately after doing this so no one walking by can see the password you just entered.

  2) Configure your shell history system to ignore commands beginning with `export TSSHBATCH`. That way your plain text password will never appear in the shell command history.

3) Make sure you don't leave a logged in session unlocked so that other users could walk up and see your password by displaying the environment.

This approach is best when you want your login credentials available for the duration of an *entire login session.*

- Store your password in an encrypted file and decrypt it inline.

  First, you have to store your password in an encrypted format. There are several ways to do this, but `gpg` is commonly used:

  ```
  echo "my_weak_pw" | gpg -c >mysecretpw
  ```

  Provide a decrypt passphrase, and you're done.

  Now, you can use this by decrypting it inline as needed:

  ```
  #!/bin/sh
  # A demo scripted use of tsshbatch with CLI password passing

  MYPW=‘cat mysecretpw | gpg‘   # User will be prompted for unlock passphrase

  tsshbatch.py -n joe.luser -p $MYPW hostlist1 command1 arg
  tsshbatch.py -n joe.luser -p $MYPW hostlist2 command2 arg
  tsshbatch.py -n joe.luser -p $MYPW hostlist3 command3 arg
  ```

  This approach is best when you want your login credentials available for the duration of *the execution of a script.* It does require the user to type in a passphrase to unlock the encrypted password file, but your plain text password never appears in the wild.

Authentication Using Key Exchange

For most applications of `tsshbatch`, it is much simpler to use key-based authentication. For this to work, you must first have pushed ssh keys to all your hosts. You then instruct `tsshbatch` to use key-based authentication rather than name and password. Not only does this eliminate the need to constantly provide name and password, it also eliminates passing a plain text password on the command line and is thus far more secure. This also overcomes the problem of having different name/password credentials on different hosts.

By default, `tsshbatch` will prompt for name and password if they are not provided on the command line. To force key- authentication, use the `-k` option:

```
tsshbatch.py -k AIX-prod-hosts ls -al
```

This is so common that you may want to set it in your `$TSSHBATCH` environment variable so that keys are used by default. If you do this, there may still be times when you want for force prompting for passwords rather than using keys. You can do this with the `-K` option which effectively overrides any prior `-k` selection.

Executing A `sudo` Command

`tsshbatch` is smart enough to handle commands that begin with the `sudo` command. It knows that such commands *require* a password no matter how you initially authenticate to get into the system. If you provide a password - either via interactive entry or the `-p` option - by default, `tsshbatch` will use that same password for `sudo` promotion.

If you provide no password - you're using `-k` and have not provided a password via `-p` - `tsshbatch` will prompt you for the password `sudo` should use.

You can force `tsshbatch` to ask you for a `sudo` password with the `-S` option. This allows you to have one password for initial login, and a different one for `sudo` promotion.

Any time you a prompted for a `sudo` password and a login password has been provided (interactive or `-p`), you can accept this as the `sudo` password by just hitting `Enter`.

> **Note**
>
> `tsshbatch` makes a reasonable effort to scan your command line and/or command file contents to spot explicit invocations of the form `sudo ...`. It will ignore these if they are inside single- or double quoted strings, on the assumption that you're quoting the literal string `sudo ...` for some other purpose.
> However, this is not perfect because it is not a full reimplementation of the shell quoting and aliasing features. For example, if you invoke an alias on the remote machine that resolves to a `sudo` command, or you run a script with a `sudo` command in it, `tsshbatch` has no way to determine what you're trying to do. For complex applications, it's best to write a true shell script, push it all the machines in question via `-P`, and then have `tsshbatch` remotely invoke it with `sudo myscript` or something similar.
> As always, the best way to figure out what the program thinks you're asking for is to run it in test mode and look at the diagnostic output.

Precedence Of Authentication Options

`tsshbatch` supports these various authentication options in a particular heirarchy using a "first match wins" scheme. From highest to lowest, the precedence is:

1. Key exchange
2. Forced prompting for name via -N. Notice this cancels any previously requested key exchange authentication.
3. Command Line/$TSSHBATCH environment variable sets name
4. Name picked up from $USER (Default behavior)

If you try to use Key Exchange and `tsshbatch` detects a command beginning with `sudo`, it will prompt you for a password anyway. This is because `sudo` requires a password to promote privilege.

File Transfers

The `-G` and `-P` options specify file `GET` and `PUT` respectively. Both are followed by a quoted file transfer specification in the form:

```
"path-to-source-file path-to-destination-directory"
```

Note that this means the file will always be stored under its original name in the destination directory. *Renaming isn't possible during file transfer.*

However, `tsshbatch` always does `GETs` then `PUTs` *then* any outstanding command (if any) at the end of the command line. This permits things like renaming on the remote machine after a `PUT`:

```
tsshbatch.py -P "foo ./" hostlist mv -v foo foo.has.a.new.name
```

`GETs` are a bit of a different story because you are retrieving a file of the same name on every host. To avoid having all but the last one clobber the previous one, `tsshbatch` makes forces the files you `GET` to be uniquely named by prepending the hostname and a `"-"` to the actual file name:

```
tsshbatch.py -H myhost -G "foo ./"
```

This saves the file `myhost-foo` in the `./` on your a local machine.

These commands do not recognize any special directory shortcut symbols like `~/` like the shell interpreter might. You must name file and directory locations using ordinary pathing conventions. You can put as many of these requests on the command line as you like to enable `GETs` and `PUTs` of multiple files. You cannot, however, use filename wildcards to specify multi-file operations.

You can put multiple `GETs` or `PUTs` on the command line for the same file. They do not override each other but are *cummulative*. So this:

```
tsshbatch.py -P"foo ./" -P"foo /tmp" ...
```

Would put local file `foo` in both `./` and `/tmp` on each host specified. Similarly, you can specify multiple files to `GET` from remote hosts and place them in the same local directory:

```
tsshbatch.py -G"/etc/fstab ./tmp" -G"/etc/rc.conf ./tmp" ...
```

You may also put file transfer specifications into a `cmdfile` via the `.getfile` and `.putfile` directives. This is handy when you have many to do and don't want to clutter up the command line. Each must be on its own line in the `cmdfile` and in the same form as if it were provided on the command line:

```
.getfile /path/to/srcfile destdir    # This will get a file
.putfile /path/to/srcfile destdir    # This will put a file
```

File transfers are done in the order they appear. For instance, if you have a file transfer specification on the command line and then make reference to a `cmdfile` with a file transfer specification in it, the one on the command line gets done first.

> **Note**
>
> Keep in mind that `tsshbatch` always processes file transfers *before* executing any commands, no matter what order they appear in the `cmdfile`. If you have this in a `cmdfile`:
>
> ```
> echo "Test"
> .putfile "./myfile /foo/bar/baz/"
> ```
>
> The file will be transferred *before* the `echo` command gets run. This can be counterintuitive. It's therefore recommended that you put your file transfers into a single file, and `.include` it as the first thing in your `cmdfile` to make it obvious that these will be run first.

By default, `tsshbatch` aborts if any file transfer fails. This is unlike the case of failed commands which are reported but do *not* abort the program. The rationale' for this is that you may be doing both file transfer and command execution with a single `tsshbatch` invocation, and the commands may depend on a file being transfered first.

If you are sure no such problem exists, you can use the `-a` option to disable abort-after-failure semantics on file transfer. In this case, file transfer errors will be reported, but `tsshbatch` will continue on to the next transfer request.

`tsshbatch` does preserve permissions when transferring files. Obviously, for this to work, the destination has to be writable by the ID you're logging in with.

> **Note**
>
> The file transfer logic cannot cope with filenames that contain spaces. The workaround is to either temporarily rename them, or put them in a container like a tarball or zip file and transfer that instead.

Commenting

Both the `cmdfile` and `hostlistfile` can be freely commented using the `#` character. Everything from that character to the end of that line is ignored. Similarly, you can use whitespace freely, except in cases where it would change the syntax of a command or host name.

Includes

You may also include other files as you wish with the `.include filename` directive anywhere in the `cmdfile` or `hostlistfile`. This is useful for breaking up long lists of things into smaller parts. For example, suppose you have three host lists, one for each major production areas of your network:

```
hosts-development
hosts-stage
host-production
```

You might typically run different `tsshbatch` jobs on each of these sets of hosts. But suppose you now want to run a job on all of them. Instead of copying them all into a master file (which would be instantly obsolete if you changed anything in one of the above files), you could create `hosts-all` with this content:

```
.include hosts-development
.include hosts-stage
.include hosts-production
```

that way if you edited any of the underlying files, the `hosts-all` would reflect the change.

Similarly you can do the same thing with the `cmdfile` to group similar commands into separate files and include them.

`tsshbatch` does not enforce a limit on how deeply nested `.includes` can be. An included file can include another file and so on. However, if a circular include is detected, the program will notify you and abort. This happens if, say, file1 includes file2, file2 includes file3, and file3 includes file1. This would create an infinite loop of includes if permitted. You can, of course, include the same file multiple times, either in a single file or throughout other included files, so long as no circular include is created.

Search Paths

`tsshbatch` supports the ablity to search paths to find files you've referenced. The search path for `cmdfiles` is specified in the `$TSSHBATCHCMDS` environment variable. The `hostlistfiles` search path is specified in the `$TSSHBATCHHOSTS` environment variable. These are both in standard path delimited format for your operating system. For example, on Unix-like systems these look like this:

```
export TSSHBATCHCMDS="/usr/local/etc/.tsshbatch/commands:/home/me/.tsshbatch/commands"
```

And so forth.

These paths are honored both for any files you specify on the command line as well as for any files you reference in a `.include` directive. This allows you to maintain libraries of standard commands and host lists in well known locations and `.include` the ones you need.

`tsshbatch` will always first check to see if a file you've specified is in your local (invoking) directory and/or whether it is a fully qualified file name before attempting to look down a search path. If a file exist in several locations, the first instance found "wins". So, for instance, if you have a file called `myhosts` somewhere in the path defined in `$TSSHBATCHHOSTS`, you can override it by creating a file of same name in your current working directory.

`tsshbatch` also checks for so-called "circular includes" which would cause an infinite inclusion loop. It will abort upon discovering this, prior to any file transfers or commands being executed.

An Overview Of Variables

As you become more sophisticated in your use of `tsshbatch`, you'll begin to see the same patterns of use over and over again. Variables are a way for you to use "shortcuts" to reference long strings without having to type the whole string in every time. So, for example, instead of having to type in a command like this:

```
myfinecommand -X -Y -x because this is a really long string
```

You can just define variable like this:

```
.define __MYCMD__ =  myfinecommand -X -Y -x because this is a really long string
```

From then on, instead of typing in that long command on the command line or in a command file, you can just use `__MYCMD__` and `tsshbatch` will substitute the string as you defined it whenever it encounters the variable.

Variables can be used pretty much everwhere:

- In `hostlistfiles` or in the hostnames listed with `-H`:

  ```
  .define __MYDOMAIN__ = stage.mydomain.com
  #.define __MYDOMAIN__ = prod.mydomain.com

  host1.__MYDOMAIN__
  host2.__MYDOMAIN__
  ```

  Now you can switch `tsshbatch` operation from stage to prod simply by changing what is commented out at the beginning.

- In file transfer specifications:

  ```
  tsshbatch.py -xP"./fstab-__MYHOSTNAME__  ./" hostlist
  tsshbatch.py -xG"/etc/__OSNAME__-release ./" hostlist
  ```

- In `cmdfiles`:

  ```
  .define __SHELL__ = /usr/local/bin/bash

  __SHELL__ -c myfinescript
  ```

Note

A variable can have pretty much any name you like excepting the use of metacharacters like `<` or `!`. But if you are not careful, you can cause unintended errors:

```
.define foo = Slop

myfoodserver.foods.com
```

When you run `tsshbatch` it will then turn the server name into `mySlopdserver.Slopds.com` - probably not what you want.
So, it's a Really Good Idea (tm) to use some kind of naming scheme to make variables names stand out and make them unlikely to conflict accidentally with command- and host strings.

Types Of Variables

`tsshbatch` has three different kinds of variables:

- *User Defined Variables* are the kind in the example above. You, the user, define them as you wish in a `cmdfile` or `hostlistfile`.
- *Execution Variables* run any program or script of your choosing (on the same machine you're running `tsshbatch`) and assign the results to a variable.
- *Builtin Variables* are variables the `tsshbatch` itself defines. You can override their default values by creating a User Defined Variable of the same name.

Where And When Do Variables Get Processed?

User Defined and Execution Variables are defined in either a `hostlistfile` or `cmdfile`.

Builtin Variables are defined within `tsshbatch` itself unless you override them.

User Defined Variables are *all* read in and *then* used. If you do something like this:

```
.define __FOO__ = firstfoo
echo __FOO__
.define __FOO__ = secondfoo
```

You'll get an output of ... `secondfoo`! Why? Because before `tsshbatch` tries to run anything, it has to process all the `cmdfiles`, `hostlistfile`, and the command line content. So, before we ever get around to doing an `echo __FOO__` on some host, the second definition of __FOO__ has been read in ... and last definition wins.

Execution Variables are like User Defined Variables. They get processed a single time *at the time they're read in* from a `cmdfile` or `hostlistfile`.

Builtin Variables get evaluated *every time "tsshbatch" prepares to connect to a new host* (unless you've overriden them). That way, the most current value for them is available for use on the next host.

Keep in mind that `tsshbatch` isn't a programming language. It's "variables" are simple string substitutions with "last one wins" semantics. There is no notion of scope, for example. If you define the same variable in, say, a `cmdfile` and also in the `hostlistfile`, the latter will "win". Why? Because `hostlistfiles` are always read in after any `cmdfiles`.

Finally, variable references in a definition are *ignored*. Say you do this in a `cmdfile`:

```
.define __CLEVER      __ = __REALLYCLEVER__
.define __REALLYCLEVER__ = Not That Smart
echo __CLEVER__
```

You will get this output, `__REALLYCLEVER__`! Why? Because, the variable references on the right side of a definition statement are never replaced. This is a concious design choice to keep variable definition and use as simple and obvious as possible. Allowing such "indirect" definitions opens up a treasure trove of maintenance pain you really want to avoid. Trust us on this one.

User-Defined Variables

`tsshbatch` allows you to define variables which will then be used to replace matching strings in `cmdfiles`, `hostlistfiles`, and file transfer specifications. For example, suppose you have this in a `hostlistfile`:

```
.define DOMAIN=.my.own.domain.com

host1DOMAIN
host2DOMAIN
host3DOMAIN
```

At runtime, the program will actually connect to `host1.my.own.domain.com`, `host2.my.domain.com`, and so on. This allows for ease of modularization and maintenance of your files.

Similarly, you might want define `MYCMD=some_long_string` so you don't have to type `some_long_string` over and over again in a `cmdfile`.

There are some "gotchas" to this:

- The general form of a variable definition is:

  ```
  .define name = value
  ```

  You have to have a name but the value is optional. `.define FOO=` simply replaces any subsequent `FOO` strings with nothing, effectively removing them.

  Any `=` symbols to the right of the one right after `name` are just considered part of the variable's value.

Whitespace around the = symbol is optional but allowed.

- Variables are substituted in the order they appear:

```
.define LS = ls -alr
LS /etc               # ls -alr /etc
.define LS = ls -1
LS /foo               # ls -1 /foo
```

- Variable names and values are *case sensitive*.
- Variables may be defined in either `cmdfiles` or `hostlistfiles` but they are *visible to any subsequent file that gets read*. For instance, `cmdfiles` are read before any `hostlistfiles`. Any variables you've defined in a `cmdfile` that happen to match a string in one of your hostnames will be substituted.

  This is usually not what you want, so be careful. One way to manage this is to use variables names that are highly unlikely to ever show up in a hostname or command. That way your commands and hostnames will not accidentally get substrings replaced with variable values. For example, you might use variable names like `--MYLSCOMMAND--` or `__DISPLAY_VGS__`.
- Variable substitution is also performed on any host names or commands passed on the command line.

Execution Variables

Execution Variables are actually a special case of User Defined Variables. That is, they are evaluated at the same time and in the same manner as any other User Defined Variable. The difference is that a User Defined Variable describes a *literal string replacement*. But an Execution Variable *runs a command, program, or script and assigns the results to the variable*.

For example, suppose you want create a file on many machines, and you want that file to be named based on who ran the `tsshbatch` job. You might do this in a `cmdfile`:

```
.define __WHOAMI__ = ! whoami
touch __WHOAMI__-Put_This_Here.txt
```

So, if ID `luser` is running `tsshbatch`, a file called `luser-Put_This_Here.txt` will be created (or have its timestamp updated) on every machine in the `hostlistfile` or named with `-H`.

Notice it is the `!` character that distinguishes an Execution Variable from a User Defined Variable. It is this character that tells `tsshbatch`, "Go run the command to the right of me and return the results." The trailing space is optional and the definition could be written as:

```
.define __WHOAMI__ = !whoami
```

If the command you specify returns multiple lines of output, it's up to you to process it properly. `tsshbatch` does no newline stripping or other postprocessing of the command results. This can make the output really "noisy". `tssbatch` normally reports a summary of the command and its results. But if you do something like this:

```
.define __LS__ = ! ls -al
echo __LS__
```

You will get a multiline summary of the command and then the actual output - which is also multiline. This gets to be obnonxious pretty quickly. You can make a lot of this go away with the `-q`, or "quiet" option.

Note

It's important to remember that the program you are invoking *runs on the same machine as* `tsshbatch` *itself*, NOT each host you are sending commands to. In other words, just like Builtin Variables, Execution Variables are *locally defined*.

Builtin Variables

As noted previously, Builtin Variables are created by `tsshbatch` itself. They are created for each new host connection so that things like time, host number, and hostname are up-to-date.

As of this release, `tsshbatch` supports the following Builtins:

`__DATE__`	Date in YYYYMMDD format
`__DATETIME__`	Date and time in YYYYMMDDHHMMSS format
`__HOSTNAME__`	Full name of current host as passed to `tsshbatch`
`__HOSTNUM__`	Count of host being processed, starting at 1
`__HOSTSHORT__`	Leftmost component of hostname as passed to `tsshbatch`
`__LOGINNAME__`	User name used for remote login. For key auth, name of tsshbatch user.
`__TIME__`	Time in HHMMSS format

Using Builtin Variables

There are times when it's convenient to be able to embed the name of the current host in either a command or in a file transfer specification. For example, suppose you want to use a single invocation of `tsshbatch` to transfer files in a host-specific way. You might name your files like this:

```
myfile.host1
myfile.host2
```

Now, all you have to do is this:

```
tsshbatch.py -xH "host 1 host2" -P "myfile.__HOSTNAME__ ./"
```

When run, `tsshbatch` will substitute the name of the current host in place of the string `__HOSTNAME__`. (*Note that these are **double** underbars on each side of the string.*)

You can do this in commands (and commands within command files) as well:

```
tsshbatch.py -x hosts 'echo I am running on __HOSTNAME__'
```

Be careful to escape and quote things properly, especially from the the command line, since < and > are recognized by the shell as metacharacters.

There are two forms of host name substitution possible. The first, `__HOSTNAME__` will use the name *as you provided it*, either as an argument to `-H` or from within a host file.

The second, `__HOSTSHORT__`, will only use the portion of the name string you provided up to the leftmost period.

So, if you specify `myhost1.frumious.edu`, `__HOSTNAME__` will be replaced with that entire string, and `__HOSTSHORT__` will be replaced by just `myhost1`.

Notice that, in no case does `tsshbatch` do any DNS lookups to figure this stuff out. It just manipulates the strings you provide as hostnames.

The symbols `__HOSTNAME__` and `__HOSTSHORT__` are like any other symbol you might have specified yourself with `.define`. *This means you can override their meaning.* For instance, say you're doing this:

```
tsshbatch.py -x myhosts  echo "It is: __HOSTNAME__"
```

As you would expect, the program will log into that host, echo the hostname and exit. But suppose you don't want it to echo something else for whatever reason. You'd create a command file with this entry:

```
.define __HOSTNAME__ = Really A Different Name
```

Now, when you run the command above, the output is:

```
It is: Really A Different Name
```

In other words, `.define` has a *higher precedence* than the preconfigured values of `HOSTNAME` and `HOSTSHORT`.

Noise Levels

`tsshbatch` defaults to a medium level of reporting as it runs. This includes connection reporting, headers describing the command being run on every host,and the results written to `stdin` and `stdout`. Each line of reporting output begins with `--->` to help you parse through the output if you happen to be writing a program that post-processes the results from `tsshbatch`.

This output "noise" is judged to be right for most applications of the program. There are times, however, when you want more- or less "noise" in the output. There are several `tsshbatch` options that support this.

These options *only affect reporting of commands you're running.* They do not change the output of file transfer operations. They also do not change error reporting, which is always the same irrespective of current noise level setting.

`-q` or "quiet" mode, reduces the amount of output noise in two ways. First, it silences reporting each time a successful connection is made to a host. Secondly, the command being run isn't reported in the header. For example, normally, running `ls -l` is reported like this:

```
--->  myhost:    SUCCESS: Connection Established
--->  myhost (stdout) [ls -l]:
...
--->  myhost (stderr) [ls -l]:
```

In quiet mode, reporting looks like this:

```
--->  localhost (stdout):
...
--->  localhost (stderr):
```

The main reason for this is that some commands can be very long. With execution variables, it's possible to create commands that span many lines. The quiet option gives you the ability to suppress echoing these long commands for each and every host in your list.

`-y` or "noisy" mode, produces normal output noise but also replicates the hostname and command string *for every line of output produced.* For instance, `ls -1` might normally produce this:

```
--->  myhost:    SUCCESS: Connection Established
--->  myhost (stdout) [ls -1]:

       backups
       bin
```

But in noisy mode, you see this:

```
--->  myhost:    SUCCESS: Connection Established
```

```
[myhost (stdout) [ls -1]]        backups
[myhost (stdout) [ls -1]]        bin
```

Again, the purpose here is to support post-processing where you might want to search through a large amount of output looking only for results from particular hosts or commands.

`-s` or "silent" mode returns *only the results from running the commands*. No headers or descriptive information are produced. It's more-or-less what you'd see if you logged into the host and ran the command interactively. For instance, `ls -l` might look like this:

```
total 44
drwxr-xr-x  2 splot splot 4096 Nov  5 14:54 Desktop
drwxrwxr-x 39 splot splot 4096 Sep  9 14:57 Dev
drwxr-xr-x  3 splot splot 4096 Jun 14  2012 Documents
```

The idea here is to use silent mode with the various variables described previously to customize your own reporting output. Imagine you have this in a `cmdfile` and you run `tsshbatch` in silent mode:

```
.define __USER__ = ! echo $USER
echo "Run on __HOSTNAME__ on __DATE__ at __TIME__ by __USER__"
uname -a
```

You'd see output along these lines:

```
Run on myhost on 20991208 at 141659 by splot
Linux myhost 3.11.0-12-generic #19-Ubuntu SMP Wed Oct 9 16:20:46 UTC 2013 x86_64 x86_64 x86_64 G
```

OTHER

- Comments can go anywhere.
- Directives like `.define` and `.include` must be the first non-whitespace text on the left end of a line. If you do this in a `cmdfile`:

  ```
  foo .include bar
  ```

  `tsshbatch` thinks you want to run the command `foo` with an argument of `.include bar`. If you do it in a `hostlistfile`, the program thinks you're trying to contact a host called `foo .include bar`. In neither case is this likely to be quite what you had in mind. Similarly, everything to the right of the directive is considered its argument (up to any comment character).
- Whitespace is not significant at the beginning or end of a line but it is preserved within `.define` and `.include` directive arguments as well as within commmand definitions.
- Strictly speaking, you do not have to have whitespace after a directive. This is recognized:

  ```
  .includesomefileofmine
  .definemyvar=foo
  ```

  But this is *strongly* discouraged because it's really hard to read.
- `tsshbatch` writes the `stdout` of the remote host(s) to `stdout` on the local machine. It similarly writes remote `stderr` output to the local machine's `stderr`. If you wish to suppress `stderr` output, either redirect it on your local command line or use the `-e` option to turn it off entirely. If you want everything to go to your local `stdout`, use the `-E` option.
- You must have a reasonably current version of Python 2.x installed. It almost certainly will not work on Python 3.x because it uses the deprecated `commands` module. This decision was made to make the program as backward compatible with older versions of Python as possible (there is way more 2.x around than there is 3.x).
- If your Python installation does not install `paramiko` you'll have to install it manually, since `tsshbatch` requires these libraries as well.

- `tsshbatch` has been run extensively from Unix-like systems (Linux, FreeBSD) and has had no testing whatsoever on Microsoft Windows. If you have experience using it on Windows, do please share with the class using the email address below. While we do not officially support this tool on Windows, if the changes needed to make it work properly are small enough, we'd consider updating the code accordingly.

BUGS AND MISFEATURES

- You will not be able to run remote `sudo` commands if the host in question enables the `Defaults requiretty` in its `sudoers` configuration. Some overzealous InfoSec folks seem to think this is a brilliant way to secure your system (they're wrong) and there's nothing `tsshbatch` can do about it.
- When `sudo` is presented a bad password, it ordinarily prints a string indicating something is wrong. `tsshbatch` looks for this to let you know that you've got a problem and then terminates further operation. This is so that you do not attempt to log in with a bad password across all the hosts you have targeted. (Many enterprises have policies to lock out a user ID after some small number of failed login/access attempts.)

  However, some older versions of `sudo` (noted on a RHEL 4 host running `sudo` 1.6.7p5) do not return any feedback when presented with a bad password. This means that `tsshbatch` cannot tell the difference between a successful `sudo` and a system waiting for you to reenter a proper password. In this situation, if you enter a bad password, the *the program will hang.* Why? `tsshbatch` thinks nothing is wrong and waits for the `sudo` command to complete. At the same time, `sudo` itself is waiting for an updated password. In this case, you have to kill `tsshbatch` and start over. This typically requires you to put the program in background (`‘Ctrl-Z` in most shells) and then killing that job from the command line.

  There is no known workaround for this problem.

OTHER, SIMILAR PRODUCTS

It's always interesting to see how other people approach the same problem. If you're interested in this general area of IT automation, you may want to also look at `Ansible`, `Capistrano`, `Cluster SSH`, `Fabric`, `Func`, and `Rundeck`.

COPYRIGHT AND LICENSING

AUTHOR

`Tim Daneliuk`
`tsshbatch@tundraware.com`

DOCUMENT REVISION INFORMATION

`$Id: tsshbatch.rst,v 1.174 2016/01/19 00:10:22 tundra Exp $`

This document was produced with `emacs`, `RestructuredText`, and `TeX Live`.

You can find the latest version of this program at:

http://www.tundraware.com/Software/tsshbatch