

TP2 Packaging Administration Guide

This document covers the use of TP2 to manage packaged software. It covers installation, upgrade and removal of packages. It covers management of security, auditing, verification of software, as well as repository management and recommended update policies.

Version	Date	Author	Changes
1.0	November 2007	Simon Edwards	Original version
1.1	October 2009	Simon Edwards	Various updates and improvements.

Table of Contents

1 Purpose of Document.....	4
2 Introduction to Packaging.....	4
2.1 Summary of Features TP2 Offers.....	4
3 Understanding Namespaces.....	6
3.1 What are Namespaces?.....	6
3.2 Benefits of Namespaces.....	6
3.3 Listing Available Namespaces.....	6
3.4 Creating a New Namespace.....	7
3.5 Removing an Existing Namespace.....	7
4 Package Installations.....	9
4.1 Preview Installations.....	9
4.1.1 Steps not performed by Preview Installs.....	10
4.2 Real Installations.....	11
4.3 Common Installation Issues.....	11
5 Package Upgrades.....	13
5.1 Install vs Upgrade – the differences.....	13
5.2 Upgrading to Specific Versions.....	15
5.3 Cross-Repository Searching.....	15
5.4 Package Re-installation and Downgrading.....	16
6 Package Removals.....	17
6.1 Preview Removals.....	17
6.2 Removing Packages with Dependencies.....	19
6.3 Non-standard Package Removal Scenarios.....	20
7 Working with Bundles.....	23
7.1 What are Bundles?.....	23
7.2 Listing available Bundles in Repository.....	23
7.3 Bundle Installation Example.....	24
7.4 Affect on Package Listings.....	25
7.5 Partial Bundle Removal.....	26
7.6 De-bundling.....	28
7.7 Making up a bundle.....	29
7.8 Bundle Removal.....	30
7.9 Updating Bundles.....	30
7.10 Limitations and Future Improvements.....	30
8 Repository Management.....	32
8.1 Default Repositories.....	32
8.2 Listing Repository Contents.....	33
8.3 Retrieving More Repository Information.....	34
8.4 Managing Repository Contents.....	36
8.5 Automatic Repository Management.....	37
9 Package Cleaning and Post-Boot Actions.....	38
9.1 When do Packages need Cleaning?.....	38
9.2 How to Clean Packages.....	38
10 Repackaging.....	41
10.1 What is Repackaging.....	41
10.2 When is repackaging typically used.....	41
10.3 Examples.....	41
10.4 Where are the Packages?.....	43
10.5 Restoring a Re-packaging version.....	44
11 Log File Management.....	45
11.1 What is logged?.....	45

- 11.2 Examining Logs.....46
- 11.3 Managing Log Space.....47
- 12 Namespace and Package Security.....49
 - 12.1 Basic UNIX-level security considerations.....49
 - 12.2 Package Signing Concepts.....50
 - 12.3 Generating a Public/Private Key Pair.....50
 - 12.4 Signing a Package.....51
- 13 TP2 Security and Auditing Daemon.....52
 - 13.1 Purpose of Daemon.....52
 - 13.2 General Configuration and Start-up.....52
 - 13.2.1 Entries available for "allow_namespace_changes".....53
 - 13.3 User Configuration Steps.....54
 - 13.4 Available User-available Functionality.....54
- 14 General Namespace Administration Topics.....55
 - 14.1 Namespace Verification.....55
 - 14.2 Namespace Permissions Fixing.....56
 - 14.3 First Boot Scripts.....56
- Appendix A: Namespace Configuration File Options.....58
- Appendix B: Automated File System Support Configurations.....60

1 Purpose of Document

TP2 has been written to meet several requirements – one of which is ease of use. This guide has been written for users wishing to install and manage software packages using TP2. It does not contain installation instructions for TP2 itself – for those see the “TP2 Installation Guide”. This guide covers all details you need to manage TP2 in its entirety – not just installing packages, but upgrading them, removing them, verifying installations, cleaning up after failed package installs, as well as managing repositories and namespaces.

This guide does not cover any particular UNIX-like OS – TP2 was written to work across all UNIX-like environments, and is in use on Linux, Solaris and HP-UX amongst other environments.

2 Introduction to Packaging

Packaging solutions are software that allows users and administrators to easily install, remove and upgrade collections of files that are used to provide a software service (such as a word processor, or a 'C' compiler, for example). Many of the packaging solutions that are currently available on the market allow for installation from the command line or GUI interface. They also allow installation from remote sources over the Internet and can handle dependencies – when the package requires other packages to function correctly.

TP2 is no exception – it offers a command line, a text console based interface and a GUI to install software. However TP2 offers a unique feature compared to most packaging solutions – namespaces.

Namespaces allow software to be deployed into different areas on a machine – in fact they allow different users to manage their own packages. A common use might be for individual users to have a namespace in their own directories. This would then allow different users to manage packages for themselves. More details on name spaces, how they work and are managed can be found later in this guide.

2.1 Summary of Features TP2 Offers

TP2 offers many features whilst remaining easy to use and easy to manage. The features include:

- *Namespace Support* – Allows the same package (even at the same level) to be installed into multiple locations on the host machine if desired.
- *Full Dependency Management* – it is possible to define that one package relies on another being installed first. In such cases TP2 will pull in all such dependencies during installation, or will abort if any dependency can not be found.
- *Multiple Repository Support* – Files can be retrieved from different locations to meet dependencies. The locations are defined by the administrator, and can be local directories, FTP accounts or HTTP URL's.
- *Atomic Package Installs* – a package will install or it will fail. If it fails to install any existing programs that were “overwritten” will be restored.

- *Intelligent File Installs* – Files are only replaced if they differ from an existing copy. This means that installations of existing programs are less likely to impact running software allowing most, if not all, package installations to be performed “on-line”.
- *Full cross-package Integrity* – meta information regarding each and every file a package installed is kept. The toolset logically handles the situation when multiple packages install the same file.
- *Package Integrity* - every file installed for a package has its configuration and contents validated during installation. A verification tool can quickly ascertain any changes to an installed package to alert administrators to potential problems.
- *Ease of Use* – a simple command line interface is available, whilst text console and GUI interfaces for package addition and removal are also available.
- *Cross Platform* – TP2 has not been designed for a particular UNIX OS – it has been designed to work on *any* UNIX-like Operating System.
- *Preview Support* - it is possible to perform software installs, upgrades, downgrades or removals in a “preview” mode to see the impact of the action before actually doing it.
- *Auditing* - An optional daemon allows generation of a central audit log on a machine covering all changes to all namespaces.
- *Non-root user support* - it is possible to define users as having abilities to perform actions that normally only root would be able to perform (such as changing namespaces).
- *Software Bundling* - multiple packages can be “bundled” together to allow easier installations of groups of software.
- *Signed Packages* - packages can be given DSA signatures and namespaces can be configured to except only certain signatures.
- *Repackaging Support* - optionally when installing software re-packaging can be used. This dynamically creates a package based on what is installed at the moment, meaning that an exact recovery is possible if you later wish to back-out an installation.
- *Automatic file system management* - it is possible (when running the optional daemon) to indicate that a namespace can support automatic file system growth (with a suitable volume manager and file system).

3 Understanding Namespaces

3.1 What are Namespaces?

Namespaces are different parts of the directory tree where software can be installed into. Once defined a namespace is referred to by name, rather than the directory it refers to. All software in a particular namespace will be installed under that specified directory. Namespaces must have completely separate directory spaces [apart from the "root" namespace, which uses the "/" directory].

Each namespace is owned by a particular user – the current version of TP2 only allows "root" or the specified user to install software into that namespace. For obvious security reasons new name spaces can only be configured or removed by the "root" user or those designated by "root" to have this facility available. However once a namespace has been defined for a user they are free to install packages as they see fit.

3.2 Benefits of Namespaces

Namespaces allow different versions of the same package to be installed at the same time on a machine without interfering with one another. They also allow different users to control areas of the file system independent of the root user.

When a non-root user makes use of a namespace all files that are installed are installed as the current user – not as the user that might be specified as part of the package. This is necessary to ensure that user is able to remove or modify the package files following the initial installation.

Namespaces are also useful since they allow test installation of software on the same machine as the software in question might already be installed locally on.

3.3 Listing Available Namespaces

As the "root" user it is possible to list all defined namespaces that have been configured. This is done using "tp2 list" for example:

```
# tp2 list --namespaces --all
```

The output generated is a series of columns showing the name of the namespace, the root directory, owner of that namespace and number of currently installed packages there.

Name	Root	Owner	Pkgs
root	/	root	0
env_rcs	/env/rcs	root	2
test1	/tmp/test1_root	snedward	0
test_03	/env/rawdata/test/03.00.00	envadmin	2
test_04	/env/rawdata/test/04.00.00	envadmin	0
test_05	/env/rawdata/test/05.00.00	envadmin	1

Note: Without the "--all" option the "root" user would only see the namespaces that they actually own.

If a non-root user was to attempt the command it would only list the namespaces for that user – for example running the same command as “snedward” would give:

Name	Root	Owner	Pkgs
test1	/tmp/test1_root	snedward	0

There are no limits on the number of namespaces that can be created. Note that each namespace has a separate meta-data repository.

3.4 Creating a New Namespace

Only “root” is able to create a new namespace by default. To do so use the “makens” sub-command, for example:

```
# tp2 make_ns --owner sedwards --root /home/sedwards --name sedwards --verbose
```

When run in “verbose” it provides some messages:

```
Log : Created directory /var/adm/tp2/ns/sedwards.
Log : Created directory /var/adm/tp2/ns/sedwards/meta.
Log : Created directory /var/adm/tp2/ns/sedwards/pkg.
Log : Created directory /var/adm/tp2/ns/sedwards/log.
Log : Created directory /var/adm/tp2/ns/sedwards/spool.
Log : Created "sedwards" namespace configuration file.
```

The namespace configuration file can also be altered by “root” if necessary, though the default settings are usually suitable.

3.5 Removing an Existing Namespace

Only the owner of a namespace or “root” can remove an existing namespace. Namespaces are removed using the “tp2 remove_ns” command. The time taken to remove a namespace depends on the number of packages that it currently contains. Also be aware that by default a namespace will not be removed if it still contains packages.

```
$ tp2 remove_ns --namespace test1 --verbose
Error: Namespace 'test1' contains 4 packages.
Error: This namespace removal will only work when '--force' is specified.
```

In the above case supplying the “--force” option will remove the existing packages and once completed successfully will remove the namespace. For example:

```
$ tp2 remove_ns --namespace test1 --verbose --force
Log :
Log : Attempting to remove 'fred' and dependencies...
Log :
Log : Obtaining lock[write] on test1 ... Got it.
Log : Getting dependency list for package [name=fred,version=1.1.0]... Done
[1 package].
Log :
Log : Following packages will be removed to meet dependencies for fred[1.1.0]:
Log :
Log : testpkg[1.0.0] - An example test package
```

TP2: Administrator's Guide

```
Log :
Log : Removal of package testpkg starting - please wait.
RM Dir      bin                      ##### 100% #####
Log : Removal of testpkg was successful.
Log : [Use "tp2log --namespace test1 --show remove+testpkg+1183723097" for
details]
Log :
Log : Removal of package fred starting - please wait.
RM Dir      bin                      ##### 100% #####
Log : Removal of fred was successful.
Log : [Use "tp2log --namespace test1 --show remove+fred+1183723098" for
details]
Log :
Log :
Log : Attempting to remove 'pigpkg' and dependencies...
Log :
Log : Obtaining lock[write] on test1 ... Got it.
Log : Getting dependency list for package [name=pigpkg,version=1.0.0]... Done
[0 packages].
Log : Removal of package pigpkg starting - please wait.
RM Dir      bin                      ##### 100% #####
Log : Removal of pigpkg was successful.
Log : [Use "tp2log --namespace test1 --show remove+pigpkg+1183723100" for
details]
Log :
Log :
Log : Attempting to remove 'newfred' and dependencies...
Log :
Log : Obtaining lock[write] on test1 ... Got it.
Log : Getting dependency list for package [name=newfred,version=2.1.0]...
Done [0 packages].
Log : Removal of package newfred starting - please wait.
RM Dir      bin                      ##### 100% #####
Log : Removal of newfred was successful.
Log : [Use "tp2log --namespace test1 --show remove+newfred+1183723163" for
details]
Log :
```

If you are not running as "root" then the following message will be shown at the end of a namespace removal:

```
Log :
Log : Namespace data almost completely removed! to complete the
Log : process please get 'root' to run the following command:
Log :
Log : rmdir /var/adm/tp2/ns/test1
Log :
```

4 Package Installations

TP2 package installation is very easy – often very few command line options are necessary. However the instructions here detail how “preview” installs can be performed first and if successful, the real installation can be done with more confidence.

Some details not dealt with until later in this guide which have an impact on software installations are;

- *Repository Handling* - although specification of a single repository is covered here later we learn about using different repositories and even defining whether repositories can be dynamically altered by software itself.
- *Package Signing* - how to work with the various options that use and support of DSA-signed packages bring.
- *Repackaging Support* - whether repackaging during software installations occurs automatically or just when requested; where the repackage software resides; and how to use these repackaged software packages.

4.1 Preview Installations

Performing a preview installation prior to a real installation is strongly recommended – it gives greater confidence that the package in question can be installed. A preview install will;

- Determine that all package dependencies can be met using the available repositories.
- Ensure that all check-install scripts will run correctly.
- Indicate the amount of disk space required for each package – and determine whether enough temporary space is available to perform the installation.

To perform a preview installation the “--preview” option must be specified, for example to install package “skulker2” the following preview command could be used:

```
$ tp2 install --pkg skulker2 --namespace sedwards --verbose
```

This will produce output similar to the following on the terminal.

```
Log : The following repositories will be scanned for software:
Log : * /tmp
Log :
Log : skulker2 [from /tmp, version 0.6.7]
Log :
Log : Check installed/selected compatibility ... Done.
Log : Getting skulker2 [file=skulker2+0.6.7.tp2 , version=0.6.7]
from /tmp
Warn : Unable to extract config from
/var/adm/tp2/ns/test1/spool/skulker2+0.6.7.tp2 - ignoring.
Log :
Log : All software retrieved to spool directory - checking space requirements.
Log :
Log : Installation calculates following disk space requirements:
Log :
Log : File system          Space During Install
Log : /testing/tst1         1776 Kb
Log :
Log : Namespace does not require signed packages.
Log : Installation of skulker2 starting - please wait.
Log : Package file : skulker2+0.6.7.tp2
Log : Log file      : install+skulker2+1255384356
Install ..er2/generators/invalid_user_recursive.lister ##### 100% #####
Log : Installation of skulker2 was successful.
Log : [Use "tp2log --namespace test1 --show install+skulker2+1255384356" for
details]
Log :
```

If the preview installation works, then simply remove the "--review" option and repeat the command.

4.1.1 Steps not performed by Preview Installs

Not everything is performed by preview installations, for example the following are not done:

- The installation scripts are not run
- No real file installations are performed (so the generated log contains no information on actual file system impact information).
- No file system growth (if configured and available) is performed.
- DSA signing is not enforced; though warnings might be issued.
- No actual repackaging is performed.

4.2 Real Installations

A real installation is almost identical to a preview installation – though some additional lines are shown to the terminal if the `--verbose` option is specified. The example below shows the process of installation of a package which has some dependencies that also need to be installed:

```
# tp2 install --pkg pack2 --repos /tmp --namespace test1 -verbose
```

In the above example package "pack2" is being installed from the "/tmp" repository into the "test1" namespace.

```
Log : Obtaining lock[write] on test1 ... Got it.
Log : Getting dependencies for pack2.
Log : Getting dependencies for testpkg.
Log : Getting dependencies for fred.
Log : pack2 [from /tmp, version 1.2.0]
Log :       requires testpkg,1.0.0,999999.0.0
Log :
Log : testpkg [from /tmp, version 1.0.0]
Log :       requires fred,0.9.0,999999.0.0
Log :
Log : fred [from /tmp, version 1.1.0]
Log :
Log : Check installed/selected compatibility ... Done.
```

Notice that each dependency that is met is shown – and following this separate log files will be shown for each of the installations.

4.3 Common Installation Issues

Sometimes packages can not be installed. The most common reasons are now explained.

- *Missing Dependencies* – the package, or some package that is a dependency, has been unable to find a dependency it requires for installation. The `TP2_PATH` environment variable should be changed to include further repositories, or additional repositories can be specified on the command line.
- *Failed Check-Install Script* – if this package, or a dependency has a check installation script that fails then the package installations will be aborted. The use of the `--force` option can be used to ignore this failure, or the particular reason which should be stated on the terminal should be addressed, and the installation re-attempted.
- *Interrupted Installation* – the command in question is aborted or the machine is shut down in the middle of an installation. In this case it may be necessary to clean the namespace – see the "TP2 Maintenance" section at the end of this guide for details.

- *Incorrect Checksum* – one of the validation features that the TP2 system offers is one of checksums that are automatically generated when the repository index file is built. Using this information for any package that is about to be installed a checksum for the retrieved file is compared to the one in the index - and if they differ an error similar to the following will appear:

```
Error:
Error: Repository/Spoiled package checksum mismatch for pack2:
Error: Repository checksum: 303057ac1ae985d320e60b7a7cff53ff
Error: Calculated checksum: d41d8cd98f00b204e9800998ecf8427e
```

This might indicate that the file has not been transferred correctly for some reason, (communications problem, program failure or lack of disk space being just some possible reasons). In this case the installation of the package in question will be aborted.

One other reason this might have occurred is that the repository index file has not been updated but the package has. In that case the checksum is very likely to have changed and even if transferred successfully to the spool directory it will fail to match the checksum.

- *Incompatible package prerequisite* - packages are able to indicate that they are incompatible with other packages. This means that attempted installation of a package might fail if the software it is incompatible with is already installed.
- *Incompatible Package type* - when a package is created by default it will be a "generic" package; suitable for all operating systems and machine architectures. Such packages contain high level scripts or programs rather than binary executables. However it is possible to indicate a package is defined for a particular operating system and/or machine architecture.

5 Package Upgrades

The same "tp2 install" command is used to upgrade packages, simply use the same commands as previously. As with installations if the package required is found in a repository then the highest level of that package is chosen by default – which is typically what people wish for.

If a package is already installed at the same or higher level than the latest version found in the repositories an error will be given by default and the "upgrade" not installed. It is possible to re-install the same version, or downgrade to an older version if required – see the section below.

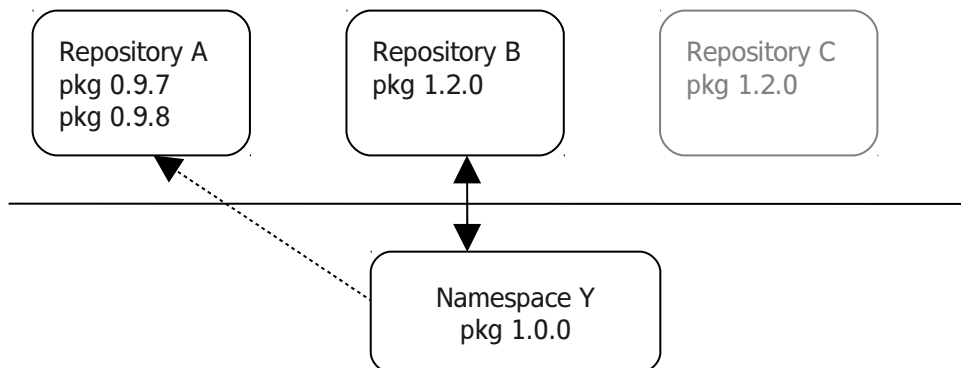
5.1 Install vs Upgrade – the differences

If a package "fred" is already installed, [at version "1.0.0"], then consider the impact of running the following command:

```
$ tp2 install --pkg fred --verbose --namespace test1
```

If "fred" was not already installed TP2 would install it from the first repository it came across. If that repository had multiple versions it would install the package with the highest version number.

When the package is already installed the action is slightly different, but logical. It will search the repositories in order and when it finds one with this package at a higher level it will install that one. If none are found at a higher level an error will be shown. An example might be:



The diagram above shows what happens when "pkg" which is installed at version "1.0.0" in a namespace is upgraded when repositories "A", "B" and "C" are defined in the repository path in that order. Notice that since a version later than the currently installed version exists in "B" it never searches through "C" (by default).

When actually performing an upgrade TP2 is intelligent in how it installs the software – it will only install the changes between the two versions of the package, not all the files again. This means that the installation for larger packages requires less disk writing, but is also not as disruptive if the package contains binaries that might still actually be running.

The other thing to take account of is that if an upgraded package does not contain some of the files of the previous revision they will be removed – which is different compared to what package managers.

An example of the log file generated by an upgrade might be:

```

Log : Arguments: --namespace=sas_rcs --pkgfile
/var/adm/tp2/ns/sas_rcs/spool/tp2+1.1.1.tp2 --verbose --progress --logfile
/var/adm/tp2/ns/sas_rcs/log/install+tp2+1253888138
Log : Namespace security checks passed.
Log : Successfully spooled package contents to /var/tmp/pkginstall.26861
Log : Existing package - implicit upgrade/downgrade will be performed.
Log : Overwriting /sas/rcs/etc/VERSION [05e17b646a817240c206186f94f8f4c70974d5dc
-> 3b1c4a149729cc044e1a39df31b3628cdf5f895]
Log : Overwriting /sas/rcs/bin/tp2install
[2ebaa1e22fc1641fffae7cb6bf1260bc63eb0f48 ->
75781a22ae2bc18b319f9d79a20bcf028073eab7]
. . .
Log : Executing post install script.
Log : Package /var/adm/tp2/ns/sas_rcs/spool/tp2+1.1.1.tp2 postinstall completed
successfully.
Log :
Log : Installation caused the following File System impact:
Log : /sas/rcs 8 Kb Increase
Log :
Log : Per file summary: Installed:0 Replaced:7 Kept:34 Removed:0
Log :
Log : Result: SUCCESS [ 0.78 secs User CPU , 0.33 secs System CPU ]
    
```

There are several points to consider when viewing the above log output:

- Each log starts with a summary of the arguments that were used during the installation. This is useful since it might indicate why certain actions described in the log are taken.
- A line indicating that the package is already installed is shown - indicating this is not a new installation.
- Each file that is changed, added or removed is shown in the log. Any files that the package does not alter compared to what is currently installed on the file system (not meta package information), is not shown.
- When files are installed or over-written the SHA1 contents of the existing file (or "0..0" for a new install) is shown, along with the SHA1 of the contents of the file being installed.
- If a script has pre or post installation scripts lines indicating these have been executed are shown.
- If any file system space was additional used, or even released, a list of affected file systems will be shown.
- A "per file summary" line is shown indicating the impact of the package; notice the "kept" count is a list of files unchanged by installation/upgrade of the package.

It is often useful to make use of the "--verbose" option when performing any action in TP2. This will log output to the screen - as shown previously - and is particularly useful if the package is large or has a large number of files or dependencies.

5.2 *Upgrading to Specific Versions*

By default when a package is upgraded it will scan each of the repositories in turn and when it finds one with a later version than the current version, the latest version from that repository will be used to perform the upgrade.

In almost all cases this will be the required action. However it is possible to specify the exact version of the package you wish to install. Consider the following command:

```
# tp2 install --pkg pkg1 --namespace test1 --version 1.2.0 --verbose
```

The above will search all configured repositories and when it finds one with version 1.2.0 of "pkg1" it will download that version and attempt the installation. If the version specified is not found then no installation will take place.

It should be noted that if the version already installed is at 1.2.0 or higher then the above command will fail – see the section on reinstallation and packaging downgrading below for more details.

The other point to take account of is that the upgrade may not upgrade to the very latest version when taking account of all repositories, since it will stop searching as soon as it finds a repository with a version greater than the currently installed version. If this is not the required action then the next section should be read!

5.3 *Cross-Repository Searching*

If you really want to upgrade to the very latest version, then a special flag should be specified as part of the installation line – the "--getlatest" flag. When this particular flag is used every repository is searched for every package, which can significantly lengthen the time for installation if a large number of repositories exist.

This additional time at least ensures that you will always get the latest version available of a package. Of course if you are using a specific version for a package install/upgrade/downgrade this option makes no difference and is quietly ignored.

5.4 Package Re-installation and Downgrading

By default if a package is already installed and the "tp2 install" command is selected again it will search for a later version to install using the repositories selected on the command line or via the "TP2PATH" environment variable.

Of course this is almost always what the user has in mind; however on occasion they may wish to re-install the current version or even down-grade to a previous version – such as a repackaged version. If this is required special command line options must be used; for reinstall use "--reinstall" whilst to downgrade to a previous version the "--downgrade" option must be specified.

Note: If downgrading is required then the "--version" option with the version in question must always be specified.

An example of performing a reinstallation might be:

```
# tp2 install --namespace testns --pkg testpkg --verbose \  
  --repos /tmp --reinstall
```

Without the --reinstall an error will be shown if the version in question is the latest available in the TP2PATH repositories:

```
Log : The following repositories will be scanned for software:  
Log : * /tmp  
Log :  
Error: Package 'testpkg' at version '3.0.0' is already installed.
```

Thus an example downgrade command might be:

```
# tp2 install --namespace testns --pkg testpkg --verbose \  
  --repos /tmp --downgrade --version 1.0.0
```

Without the explicit "--downgrade" option an error will be shown:

```
Log : The following repositories will be scanned for software:  
Log : * /tmp  
Log :  
Log :  
Log : Package 'testpkg' is already installed at a higher version [3.0.0].  
Log : Use the --downgrade option if you wish to force install the older  
version.  
Log :
```

6 Package Removals

As you might expect removing a package is as simple as package installation – simply use the “tp2 remove” command.

```
# tp2 remove --namespace test --pkg fred --verbose
```

6.1 Preview Removals

Of course one of the advantages of TP2 when installing packages is to perform a preview of the installation – and indeed the same functionality is available when packages are to be removed. As before the argument necessary to perform a preview is “--preview”. For example to remove a package called “pack2” from a namespace of “test” the following command would be used:

```
# tp2 remove --pkg pack2 --preview --namespace test --verbose
```

In this instance the output generated is as follows:

```
Log : Obtaining lock[write] on test ... Got it.
Log : Getting dependency list for package [name=pack2,version=1.2.0]... Done
[0 packages].
Log : Removal of package pack2 starting - please wait.

Log : Removal of pack2 was successful.
Log : [Use "tp2log --namespace test --show remove+pack2+1181771239" for
details]
Log :
```

The majority of the details are stored in the audited logs for the namespace though at present the log is quite empty [bug?]

```
Log : Arguments: --namespace=test1 --pkg=fred --verbose --preview --logfile
/var/adm/tp2/ns/test1/log/remove+fred+1182156721
Log : Namespace security checks passed.
Log : Updating: File 'bin/mycp' not removed - multiple owners.
Log : [Owners fred,newfred]
Log : Removing: File 'bin/mycat'.
. . .
Log : Removing: Directory 'bin'.
. . .
Warn : Unable to remove directory '/tmpmnt2/bin' - not empty or no permissions.
Log :
Log : Removal caused the following File System impact:
Log : /tmpmnt2 1480 Kb Reduction
Log :
Log : Per file summary: Removed:97
Log :
Log : Result: SUCCESS [ 0.37 secs User CPU , 0.07 secs System CPU ]
```

Once you are happy with the actions that a package removal would perform, then the true removal can take place. This is simply a matter of removing the “--preview” from the command line example shown above.

When the "--preview" option is removed the package contents will be removed, and some additional lines in the generated log might include:

```
Warn : Unable to remove directory '/tmpmnt2/bin' - not empty or no permissions.  
Log  :  
Log  : Removal caused the following File System impact:  
Log  : /tmpmnt2                               1480 Kb Reduction  
Log  :  
Log  : Per file summary: Removed:97
```

Some points to consider about the above output:

- *Failure to remove directories* - this is probably normal since if multiple packages define the same directory they will not be empty and so cannot be removed.
- *Space Reduction* - removing files will release storage - the space depending on the number and size of the files. Please note that this is just an estimation based on the block size of the file systems in question.

6.2 Removing Packages with Dependencies

Since TP2 is capable of installing packages and any necessary dependencies, the same considerations are taken into account when removing packages. Hence if you intend to remove a package which another needs [i.e. has classified it as a dependency during installation], then removal of the package will also remove any packages that depend on it!

This is another good reason for using the "--preview" option first, for example:

```
# tp2 remove --pkg testpkg --preview --namespace test --verbose
Log : Obtaining lock[write] on test ... Got it.
Log : Getting dependency list for package [name=testpkg,version=1.0.0]...
Done [1 package].
Log :
Log : Following packages will be removed to meet dependencies for
testpkg[1.0.0]:
Log :
Log :   pack2[1.2.0]                - An example test package2
Log :
Log : Removal of package pack2 starting - please wait.
Staging   bin/stuff                ##### 100% #####
Log : Removal of pack2 was successful.
Log : [Use "tp2log --namespace test --show remove+pack2+1181774207" for
details]
Log :
Log : Removal of package testpkg starting - please wait.
Staging   bin/cat                  ##### 100% #####
Log : Removal of testpkg was successful.
Log : [Use "tp2log --namespace test --show remove+testpkg+1181774207" for
details]
Log :
```

In the above case the top of the output clearly indicates that the removal of package "testpkg" will also require that "pack2" will be removed. Depending on the complexity of the dependencies the list of dependencies might be very long.

For each package that is preview to be removed a separate log file will be generated giving details of the removal process. Once you are happy that the packages can all be removed then simply remove the "--preview" option.

It should be remembered that the packages are removed as separate entities. Hence if you wished to remove package "a", but it was a dependency for package "b" which in turn was a dependency for package "c", then TP2 would remove "c", "b" and finally "a".

This is particularly important when considering the situation if the machine crashes during package removals. Each package is dealt with atomically - so when the machine boots the package removal will complete or roll-back - *but only the package that was being dealt with at the time*. Hence not all packages will be removed but the ones that remain will ensure the dependency relationships between the installed packages remain intact.

6.3 Non-standard Package Removal Scenarios

There are two scenarios that are unusual - the first is when the package removal fails since a pre-remove script specified does not work correctly.

For example consider the following output:

```
Log : Obtaining lock[write] on test1 ... Got it.
Log : Getting dependency list for package [name=pack2,version=1.1.0]... Done
[0 packages].
Log : Removal of package pack2 starting - please wait.
Error: Pre-remove for pack2 failed - aborting.
Error: Aborting after RC=1 from removal of pack2.
Error: [Use "tp2log --namespace test1 --show remove+pack2+1181834222" for
details]
```

In this case it is simply a matter of adding "--force" to the command to force the package removal. Please note that the preview option does not run the pre or post removal scripts [if a package defines them], and so this type of failure will only occur when the real package removal is attempted.

For example the command to remove the above was:

```
# tp2 remove --namespace=test1 --pkg pack2 --verbose --force
```

The output in this instance does even include reference to the problem:

```
Log : Obtaining lock[write] on test1 ... Got it.
Log : Getting dependency list for package [name=pack2,version=1.1.0]... Done
[0 packages].
Log : Removal of package pack2 starting - please wait.
RM Dir      bin                               ##### 100% #####
Log : Removal of pack2 was successful.
Log : [Use "tp2log --namespace test1 --show remove+pack2+1181834292" for
details]
Log :
```

However, viewing the detailed log would show a warning:

```
Log : Arguments: --namespace=test1 --pkg=pack2 --verbose --logfile
/var/adm/tp2/ns/test1/log/remove+pack2+1181834292 --force
Log : Namespace security checks passed.
Log : Executing Pre-remove script for pack2.
Warn : Pre-remove for pack2 failed - force continue.
Log :
Log : Per file summary: Removed:1
Log : Result: SUCCESS [ 0.38 secs User CPU , 0.34 secs System CPU ]
```

The other "problem" that might occur if packages are installed with the "loose" option is that a package removal might not actually remove all the files that appear to belong to the package itself! Consider the following output generated from "tp2 list" for two different packages:

```
# tp2 list --namespace test1 --detail pack2 testpkg
Version      : 1.1.0
```

```
Status      : Installed
Scripts     : Postinstall,Preinstall,Preremove
Altrepos    : WWW:stbe905a/tp2packages
Architecture : 9000_800
Dependencies : testpkg|1.0.0|99999999.999999.999999
Description  : An example test package2
Incompatibles : oldfred|0.0.1|99999999.999999.999999
Files       :
bin/ls      : a75124da1ecb6221976e0674e6b477fb97877685 37
bin/cat     : fb1c5c8f9be77249a0717656956fa2156df75bb2 20
```

```
# tp2 list --namespace test1 --detail testpkg
Name       : testpkg
Version    : 1.0.0
Status     : Installed
Scripts    : Checkinstall,Postinstall
Altrepos   : WWW:stbe905a/tp2packages
Dependencies : fred|0.9.0|99999999.999999.999999
Description : An example test package
Incompatibles : oldfred|0.0.1|99999999.999999.999999
Files      :
bin/ls     : a75124da1ecb6221976e0674e6b477fb97877685 37
bin2/cp2   : b56df8ed5365fca1419818aa384ba3b5e7756047 20
```

Notice that both the packages install a file called "bin/ls". Hence removal of either package will only remove a reference to "bin/ls". Only when both packages are actually removed will the file actually be removed.

This information can actually be found out using the verbose option for the namespace verification, for example:

```
# tp2 verify --namespace test1 --verbose
```

In this instance the portion of the output referring to the above is shown in bold:

```
Log : Loading configuration of packages in "test1" namespace... Done [4 packages]
Log : Checking package "fred".
Log : File overwritten by newfred: /tmp/test1_root/bin/mycp
Log : Checking package "newfred".
Log : Checking package "pack2".
Log : File overwritten by newfred: /tmp/test1_root/bin/cat
Log : Checking package "testpkg".
Log : File overwritten by pack2: /tmp/test1_root/bin/ls
```

When a preview remove is performed that information is actually shown in the log file, again highlighted in bold:

```
Log : Arguments: --namespace=test1 --pkg=testpkg --verbose --preview --logfile /var/adm/tp2/ns/test1/log/remove+testpkg+1182161822
Log : Namespace security checks passed.
Log : Updating: File 'bin/ls' not removed - multiple owners.
Log : [Owners testpkg,pack2]
Log : Removing: File 'bin2/cp2'.
Log : Removing: Directory 'bin'.
Log : Result: SUCCESS [ 0.41 secs User CPU , 0.09 secs System CPU ]
```


7 Working with Bundles

7.1 What are Bundles?

Bundles are a relatively new feature in TP2 - though as with all TP2 features the aim is to simplify software management, rather than complicate it! A bundle is simply a collection of packages. It allows a series of related software components to be installed or removed as a single item.

However unlike many package management systems bundles do not actually contain the software! Instead a bundle simply indicates the necessary software that must be installed to actually make up the bundle.

For example consider the following three "TP2" files:

```
$ ls -l /tmp/bigpkg2+1.0.0.tp2 /tmp/fred+1.1.0.tp2 /tmp/simon_bundle+1.0.0.tp2
-rw-r--r-- 1 venture users 624048 2009-10-26 10:36 /tmp/bigpkg2+1.0.0.tp2
-rw-r--r-- 1 venture users   454 2009-10-26 10:36 /tmp/fred+1.1.0.tp2
-rw-r--r-- 1 venture users   324 2009-10-26 10:36 /tmp/simon_bundle+1.0.0.tp2
```

As might be guessed from the names above the "simon_bundle" file is actually a bundle - and if it were installed it would actually install "bigpkg2" and "fred" - but notice it is actually smaller than either of the two packages.

Information on how to generate bundles is given in the "TP2 Packagers Guide".

7.2 Listing available Bundles in Repository

The first thing you might wish to do is to find the available bundles you might want to install. Bundles share the same naming schemes as packages and can be included in the directories and hence repositories.

By default when listing a repository only packages are shown, if you wish to view the bundles instead the "--bundles" argument must be provided, for example:

```
$ tp2 list --repos /tmp --bundles
Bundle      Version  Size  Description                                     Packages
=====
simon_bundle 1.0.0    8192  An example test package again                 2
```

Notice that when bundles are shown the number of packages that a bundle includes is shown. If you want more details - for example which packages that make up the bundle the format of the output can be changed, for example:

```
$ tp2 list --repos /tmp --bundles --format +packagesd
Bundle      Version  Size  Description                                     Packages
=====
simon_bundle 1.0.0    8192  An example test package again                 2

Bundle Requirements:
  bigpkg      : (Version range: 0.0.1 -> 9999999999.999999.999999)
  fred        : (Version range: 0.9.0 -> 9999999999.999999.999999)
```

The "--format" option allows the output of the repository information for packages or bundles to be altered.

7.3 Bundle Installation Example

Installation of a bundle is very easy - simply repeat a command as normal and rather than give the name of a package, simply use the bundle name instead. For example (shown broken down in sections to explain each part):

```
$ ./tp2 install --namespace testns --pkg simon_bundle --verbose --repos /tmp
Log : The following repositories will be scanned for software:
Log : * /tmp
Log :
Log : simon_bundle      [from /tmp, version 1.0.0]
Log :                   requires bigpkg,0.0.1,9999999999.999999.999999
Log :                   requires fred,0.9.0,9999999999.999999.999999
Log :
Log : bigpkg             [from /tmp, version 1.0.0]
Log :
Log : fred               [from /tmp, version 1.1.0]
Log :
Log : Check installed/selected compatibility ... Done.
Log : Getting simon_bundle [file=simon_bundle+1.0.0.tp2 , version=1.0.0]
from /tmp
Log : Getting bigpkg      [file=bigpkg+1.0.0.tp2 , version=1.0.0] from
/tmp
Log : Getting fred        [file=fred+1.1.0.tp2 , version=1.1.0] from /tmp
Log :
Log : All software retrieved to spool directory - checking space requirements.
Log :
```

The first part of the output indicates the requirements of the bundle, and then the actual versions retrieved, and from which repository.

```
Log : Installation calculates following disk space requirements:
Log :
Log : File system          Space During Install
Log : /home                688 Kb
Log :
Log : Namespace does not require signed packages.
```

Notice that the disk space is based on the amount of storage required for the largest of the packages that might need installation.

```
Log : Installation of fred starting - please wait.
Log : Package file : fred+1.1.0.tp2
Log : Log file      : install+fred+1256600758
Install  bin/mycat          ##### 100% #####
Config   bin/mycat          ##### 100% #####
Log : Installation of fred was successful.
Log : [Use "tp2log --namespace testns --show install+fred+1256600758" for details]
Log :
Log : Namespace does not require signed packages.
Log : Installation of bigpkg starting - please wait.
Log : Package file : bigpkg+1.0.0.tp2
Log : Log file      : install+bigpkg+1256600758
Install  bin/9999          ##### 100% #####
Config   bin/9999          ##### 100% #####
Log : Installation of bigpkg was successful.
```

```
Log : [Use "tp2log --namespace testns --show install+bigpkg+1256600758" for
details]
Log :
```

Notice above that each package installation is dealt with separately.

```
Log : Namespace does not require signed packages.
Log : Installation of simon_bundle starting - please wait.
Log : Package file : simon_bundle+1.0.0.tp2
Log : Log file      : install+simon_bundle+1256600763
Log : Installation of simon_bundle was successful.
Log : [Use "tp2log --namespace testns --show install+simon_bundle+1256600763" for
details]
Log :
```

Notice that the bundle is actually dealt with as a package (though it installs no software, it can actually contain pre/post install and remove scripts if necessary).

7.4 Affect on Package Listings

Once a bundle has been installed the output for it will look subtly different compared to just a namespace that has stand-alone packages, for example:

```
$ tp2 list --namespace testns
Package      Version  State    Installed  Signed by
=====
bigpkg2      1.0.0    Installed 27/10/2009
simon_bundle 1.0.0    Installed 26/10/2009
+bigpkg      1.0.0    Installed 27/10/2009
+fred        1.1.0    Installed 26/10/2009
```

A bundle is shown like a normal package, but any packages that form part of the bundle are shown underneath if indented and pre-fixed with "+". Notice that the installation dates of the packages in the bundle are given separately since the packages can still be manipulated if required.

Use of the "--detail" option is often useful with bundles, for example consider the following command:

```
$ tp2 list --namespace testns --detail simon_bundle
```

The package/bundle name has been specified to limit the output, and the "--detail" option gives:

```
Name          : simon_bundle
Version       : 1.0.0
Status        : Partial
Bundled       : bigpkg (1.0.0)
              : fred MISSING
Incompatibles : oldfred (Version range: 0.0.1 -> 9999999999.999999.999999)
Scripts       :
Text Files    :
Altrepos      : WWW:stbe905a/tp2packages
Description   : An example test package again
```

Notice that the s

7.5 Partial Bundle Removal

Since bundles are collections of packages, it is possible to manipulate the packages directly rather than the bundle still. For example, considering the above bundle shown in the previous section, it is possible to remove the package "fred" individually, for example:

```
$ tp2 remove --package fred --namespace testns
```

Now listing the namespace gives:

```
$ tp2 list --namespace testns
Package      Version    State      Installed   Signed by
=====
bigpkg2      1.0.0     Installed  27/10/2009
simon_bundle 1.0.0     Partial    26/10/2009
+bigpkg      1.0.0     Installed  27/10/2009
```

Notice that the "State" column for the bundle has been changed to "Partial" indicating the complete bundle contents are not yet installed.

Use of the "--detail" option is often useful with bundles in such circumstances, for example consider the following command:

```
$ tp2 list --namespace testns --detail simon_bundle
```

The package/bundle name has been specified to limit the output, and the "--detail" option gives:

```
Name           : simon_bundle
Version        : 1.0.0
Status         : Partial
Bundled        : bigpkg (1.0.0)
                : fred MISSING
Incompatibles  : oldfred (Version range: 0.0.1 -> 9999999999.999999.999999)
Scripts       :
Text Files     :
Altrepos       : WWW:stbe905a/tp2packages
Description    : An example test package again
```

Notice that the "Bundled" section indicates that a file is missing, hence the "partial" status of the bundle as a whole.

One further point about bundles is that they define the range of versions of the packages that they expect to be available. As seen it is possible to later install, upgrade or remove packages that form part of the bundle, though in many cases the "--force" option will be needed. However, consider the following output:

```
$ tp2 list --namespace testns2
Package      Version    State      Installed   Signed by
=====
fred         1.1.0     Installed  28/10/2009
test-bundle  2.2.0     Broken     28/10/2009
```

TP2: Administrator's Guide

```
+newfred 2.1.0 Installed 28/10/2009
+pack2 2.1.0 Installed 28/10/2009 'Simon Edwards' <simon@my.net>
+tp2 1.0.5 Installed 28/10/2009
testpkg 3.0.0 Installed 28/10/2009
```

When a bundle is broken it means that the one of the packages installed explicitly breaks the expectations of the bundle, for example in this instance:

```
$ tp2 list --namespace testns2 test-bundle --detail
Name      : test-bundle
Version   : 2.2.0
Status    : Broken
Bundled   : newfred (2.1.0)
           : pack2 (2.1.0)
           : tp2 (1.0.5) INCOMPATIBLE [1.1.0 -> 9999999999.999999.999999]
Incompatibles : oldfred [0.0.1 -> 9999999999.999999.999999]
Scripts   : Postinstall
Text Files : Copyright,Description,License,Readme
Altrepos  : WWW:stbe905a/tp2packages
Description : An example bundle
```

In the above bundle it requires the package "tp2" to be installed at versions 1.1.0 or above, and currently 1.0.5 is installed. Hence the bundle is classed as broken. Re-installation of the bundle or package will fix this particular problem.

Obviously TP2 attempts to stop you breaking bundles by checking bundle requirements as part of the incompatibilities checks that are performed when software is installed. For example, consider the following bundle that has been installed:

```
$ tp2 list --namespace testns --detail simon_bundle
Name      : simon_bundle
Version   : 1.0.0
Status    : Installed
Bundled   : bigpkg (1.0.0) [0.0.1 -> 9999999999.999999.999999]
           : fred (1.1.0) [0.9.0 -> 9999999999.999999.999999]
Incompatibles : oldfred [0.0.1 -> 9999999999.999999.999999]
Scripts    :
Text Files :
Altrepos   : WWW:stbe905a/tp2packages
Description : An example test package again
```

In the above output the version of "fred" installed is currently "1.1.0" which is within the range of versions specified by the bundle. Consider what happens when you attempt to install it:

```
$ tp2 install --namespace testns --pkg fred --repos /tmp --version 0.6.0
Error: Package 'fred' is already installed at a higher version [1.1.0].
Error: Use the --downgrade option if you wish to force install the older version.
```

The above error is a basic versioning check - TP2 typically does not expect you to install older versions of software without explicit consent. So adding the "--downgrade" option gives:

```
$ tp2 install --namespace testns --pkg fred --repos /tmp \
--version 0.6.0 --downgrade
Error: fred (0.6.0) is incompatible with bundle specifications for fred
(Version range 0.9.0 -> 9999999999.999999.999999)
```

Hence by default it will not allow you to "break" installed bundles - of course you can add the "--force" option to actually do the install and actually break it.

7.6 De-bundling

This is the name given to the process of removing just the meta-information for the bundle itself whilst not removing any software that the bundle is considered responsible for. For example consider the following configuration:

```
$ tp2 list --namespace testns
Package      Version    State      Installed   Signed by
=====
bigpkg2      1.0.0     Installed  27/10/2009
simon_bundle 1.0.0     Broken     29/10/2009
+bigpkg      1.0.0     Installed  27/10/2009
+fred        0.6.0     Installed  30/10/2009
```

The "--bundle-meta" option is used just to remove a bundle itself, rather than any software it indicates are actual dependencies or requirements for the bundle.

```
$ tp2 remove --namespace testns --verbose --bundle-meta --pkg simon_bundle
Log : Obtaining lock[write] on testns ... Got it.
Log : Removal of package simon_bundle starting - please wait.

Log : Removal of simon_bundle was successful.
Log : [Use "tp2log --namespace testns --show remove+simon_bundle+1257039628"
for details]
Log :
```

As can be see it generates little output, but afterwards viewing the namespace software gives:

```
$ tp2 list --namespace testns
Package      Version    State      Installed   Signed by
=====
bigpkg       1.0.0     Installed  27/10/2009
bigpkg2      1.0.0     Installed  27/10/2009
fred         0.6.0     Installed  30/10/2009
```

So no actual software is removed, just the bundle definition, resulting in the software that was previously considered as part of the bundle just to be seen as normal packages again.

7.7 Making up a bundle

As stated a bundle may be partially installed if it has been previously installed, but then one or more packages that are required for the bundle are removed. For example consider the following namespace status of a bundled package:

```
$ tp2 list --namespace testns simon_bundle --detail
Name           : simon_bundle
Version        : 1.0.0
Status         : Partial
Bundled        : bigpkg (1.0.0) [0.0.1 -> 9999999999.999999.999999]
                : fred MISSING
Incompatibles  : oldfred [0.0.1 -> 9999999999.999999.999999]
Scripts        :
Text Files     :
Altrepos       : WWW:stbe905a/tp2packages
Description    : An example test package again
```

So the "fred" package is missing. There are two ways in which this can be resolved, either by installing the bundle again, or by installing the package in question directly. Thus the following commands achieve the same result:

```
$ tp2 install --namespace testns --repos /tmp --pkg fred
```

```
$ tp2 install --namespace testns --pkg simon_bundle --repos /tmp --reinstall
```

If the bundle itself is only available at the same version the "--reinstall" option is obviously necessary to ensure that it passes the dependency tests.

7.8 Bundle Removal

Removal of a bundle will remove all the packages (and subsequently any dependencies that those packages have. Thus is it best to use the "--list" or "--preview" options before actually removing a bundle just to check the impact it will have:

```
$ tp2 remove --namespace testns --pkg simon_bundle --list
Log :
Log : Following packages will be removed to meet dependencies for
      simon_bundle[1.0.0]:
Log :
Log :   fred[1.1.0]           - An example fred package
Log :   bigpkg[1.0.0]        - An example test package
Log :
```

Remember that the "--bundle-meta" option as discussed in section 7.6 above (on page 28) can be used just to remove the bundle definition without impacting other packages.

However, if you are happy with it simply remove a bundle with the same command as you would remove a package. For example:

```
$ tp2 remove --namespace testns --pkg simon_bundle
```

As usual with no "--verbose" option there will be no information generated to the screen, though log files will exist for each removal, for example:

```
$ tp2 list --namespace logs testns | tail -5
Install fred          01/11/2009 23:26 No Success install+fred+1257117995
Install simon_bundle 01/11/2009 23:26 No Success install+simon_bundle+1257117995
Remove fred          01/11/2009 23:35 No Success remove+fred+1257118532
Remove simon_bundle 01/11/2009 23:35 No Success remove+simon_bundle+1257118539
Remove bigpkg        01/11/2009 23:35 No Success remove+bigpkg+1257118532
```

7.9 Updating Bundles

In this respect a bundle is similar to a package, if it is selected for installation again it will chose the latest available bundle version and install that - along with any necessary upgrades to the bundle requirements. For example consider the following two bundles:

<More XXX information here>

7.10 Limitations and Future Improvements

Bundles are still a new feature for TP2 and their handling will be improved and refined in subsequent releases. There are many aspects where

- If a depot contains a range of versions and some are incompatible with a bundle (but some are compatible) then installation of the bundle should pick the most sensible versions (probably does, but need to test it).

8 Repository Management

8.1 Default Repositories

All installations of software are sourced from "repositories". These can be specified on the command line as part of the installation to set or enhance the default repositories that can be found by referring to the environment variable "TP2PATH".

The environment variable works in a similar way to the "PATH" variable in UNIX - the elements are scanned one at a time until a suitable match is found. However unlike "PATH" the elements are separated by white space rather than colons [since this character is used to prefix the repository name with the protocol used]. For example consider the following setting:

```
# export TP2PATH="WWW:myhost.net/packages /local/packages /fred /tmp"
```

In this case each of the repositories in turn are checked, as the following example output indicates when a package installation is attempted.

```
$ tp2 install --namespace test1 --pkg fred --verbose
Log : The following repositories will be scanned for software:
Log : * WWW:www.linuxha.net/tp2repos
Log : * WWW:localhost/tp2packages
Log : * /tmp
Log :
Log : testpkg [from WWW:www.linuxha.net/tp2repos, version 3.0.0]
Log : requires fred,0.9.0,9999999999.999999.999999
Log :
Log : fred [from WWW:localhost/tp2packages, version 1.1.0]
Log :
Log : Check installed/selected compatibility ... Done.
Log : Getting testpkg [file=testpkg+3.0.0.tp2 , version=3.0.0] from
WWW:www.linuxha.net/tp2repos
Log : Getting fred [file=fred+1.1.0.tp2 , version=1.1.0] from
WWW:localhost/tp2packages
Log :
```

Notice that it although it finds "testpkg" in the first repository, it's dependency ("fred") is not there and so the next element in the "TP2PATH" is tested.

8.2 Listing Repository Contents

It is very easy to list the contents of a repository, simply use the "--repos" command line option for "tp2 list". In this instance a repository name must be given, for example:

```
$ tp2 repos --repos /tmp
```

The default format of the output is the package name, version, package size and description. For example:

Package	Version	Size	Description	Signed by
badpkg	1.0.0	24576	An example bad package	
badpkg2	1.0.0	16384	An example bad package	
bigpkg	1.0.0	671744	An example test package	
bigpkg2	1.0.0	3825664	An example test package again	
binpkg	1.0.0	8192	Package with binary	
filer	1.0.0	81920	Example filer package	
fred	0.6.0	40960	An example bad package	
fred	1.1.0	24576	An example fred package	
newfred	2.1.0	32768	A newfred package	
pack2	1.1.0	16384	An example test package2	'Simon Edwards' <x@y>
pack2	1.2.0	40960	An example test package2	
pack2	2.1.0	32768	An example test package2	
package3	2.1.0	24576	An example test package2	
package3	2.2.0	40960	An example test package2	
testpkg	1.0.0	24576	An example test package	
testpkg	3.0.0	106496	An example test package	

The contents of a repository can contain any number of different versions of the same package [as the above shows]. It is usual that each repository only contains packages that are all compatible with the same Operating system and/or architecture. In such cases the same package might reside in several repositories, though only one might contain a suitable version.

In such cases it is recommended that repositories paths include an indication on the type of contents, for example:

Repository Name	Contents
WWW:hostname/tp2packages/linux-i386	
WWW:hostname/tp2packages/hpux-generic	
WWW:hostname/tp2packages/generic	

It is in the interests of a package author to make the contents of the package "generic" if at all possible rather than constraining each package to a particular Operating system and/or architecture.

Many of the examples in this document simply reference "/tmp" or WWW:hostname/tp2packages - which in real-world multi-architecture environments may lead to problems, for example:

```
# tp2 install --namespace test1 --pkg filer --verbose --repos /tmp
```

In this case the following was shown:

```
Log : Obtaining lock[write] on test1 ... Got it.
```

```
Error: Unable to find package filer in repositories.
```

8.3 Retrieving More Repository Information

The output format given for the "tp2 list" using the "--repos" option can be modified using the optional "--format" argument. For example:

```
$ tp2 list --repos /tmp --format name,version,os,description
```

Now notice that the package in question is defined not as "generic" or suitable for HP-UX, but instead is specific to Linux or AIX for example.

Package	Version	OS	Description
badpkg	1.0.0	hpux	An example bad package
badpkg2	1.0.0	hpux	An example bad package
filer	1.0.0	linux	Example filer package
fred	1.0.0		An example fred package
fred	1.1.0	aix	An example fred package
newfred	2.1.0		A newfred package
pack2	1.1.0		An example test package2
pack2	1.2.0		An example test package2
pigpkg	1.0.0	hpux	An example bad package
testpkg	1.0.0		An example test package

The easiest way of handling this is by use of the "--compatible" option to ensure only packages that are suitable for the current machine are shown:

```
$ tp2 list --repos /tmp --compatible
```

The "--format" option used when showing the contents of the repository understands the following keywords:

Key	Displays
pkgname / name / package	The actual name of the package (which can be different from the file name prefix).
filename	The actual filename in the repository that contains this package.
version	The version of the package.
os	The operating system - blank for generic.
architecture / arch	The machine chip architecture, blank for generic.
size	The uncompressed size of the package (usually much larger than the package file itself).
description / desc	The short description of the package.
signer	Who has signed the package.

The "--format" option can be used in three different ways:

- If prefixed by a "-" the fields list are removed from the format, for example:

```
--format "-signer"
```

- If no prefix is given these fields are those exactly shown, for example:

```
--format "pkgfile size filename version"
```

- If prefixed by a "+" the fields specified are appended to the standard output, for example:

```
--format "+os,arch"
```

8.4 Managing Repository Contents

There are no in-built limits to the number of files that a repository can hold. Hence it is quite common to find lots of versions of the same packages in a repository, for example:

```
$ tp2 list --repos WWW:myhost.net/packages codemgr* --format -signer
```

Package	Version	Size	Description
codemgr2	0.1.2	335872	Codemgr2 Code Management Tools
codemgr2	0.2.2	516096	Codemgr2 Code Management Tools
codemgr2	0.2.3	524288	Codemgr2 Code Management Tools
codemgr2	0.2.4	540672	Codemgr2 Code Management Tools
codemgr2	0.2.5	540672	Codemgr2 Code Management Tools
codemgr2	0.2.7	548864	Codemgr2 Code Management Tools
codemgr2	0.3.0	557056	Codemgr2 Code Management Tools
codemgr2	0.3.1	598016	Codemgr2 Code Management Tools
codemgr2	0.3.2	598016	Codemgr2 Code Management Tools
codemgr2	0.5.0	671744	Codemgr2 Code Management Tools
codemgr2	0.5.1	720896	Codemgr2 Code Management Tools
codemgr2	0.5.2	729088	Codemgr2 Code Management Tools
codemgr2	0.5.3	729088	Codemgr2 Code Management Tools
codemgr2	0.5.4	745472	Codemgr2 Code Management Tools
codemgr2	0.5.6	753664	Codemgr2 Code Management Tools
codemgr2	0.5.7	770048	Codemgr2 Code Management Tools
codemgr2	0.5.8	811008	Codemgr2 Code Management Tools
codemgr2	0.5.9	835584	Codemgr2 Code Management Tools

This is not a problem when installing packages since TP2 does the sensible thing and chooses the one with the highest version it can find [unless the user asks for a particular version]. However to view which packages are installed, the "--latest" option can be useful to remove information on older versions of the same package in the repository.

For example:

```
$ tp2 list --repos WWW:myhost.net/packages dbmbatch_prod codemgr2 tp2 --latest
```

Here just 3 packages are shown:

Package	Version	Size	Description
codemgr2	0.9.9	1179648	Codemgr2 Code Management Tools
code_prod	2007.07.1500.02	20660224	My Batch Dev Stream 3
tp2	0.7.6	606208	Trendy Packager V2

Without the "--latest" option for example the repository returned 166 matches!

The easiest way to remove older packages from a repository is to use the "--purge" option that is available via the "tp2 repos" command. As usual this supports a preview mode to indicate the changes that will take place when the program is actually run. For example:

```
$ tp2 repos --purge --verbose --repos /tmp --keep 1 --preview
```

Notice that by default 5 versions of each package are keep. To keep less copies the "--keep" argument is specified.

In this small repository it would only remove two packages:

```
Log : Reading repository index contents ...
Log : Ignoring package 'badpkg2' [OS=hpux,Architecture=src] - only 1 packages.
Log : Ignoring package 'badpkg' [OS=hpux,Architecture=src] - only 1 packages.
Log : Ignoring package 'filer' [OS=linux,Architecture=src] - only 1 packages.
Log : Ignoring package 'newfred' [OS=Generic,Architecture=src] - only 1
packages.
Log : Ignoring package 'pigpkg' [OS=hpux,Architecture=src] - only 1 packages.
Log : Ignoring package 'testpkg' [OS=Generic,Architecture=src] - only 1
packages.
Log : Removing package 'fred' [OS=Generic,Architecture=src] Version 1.0.0.
Log : Removing package 'pack2' [OS=aix,Architecture=9000_800] Version 1.1.0.
```

Removing the "--preview" option will remove the packages specified and automatically rebuild the repository index without them

From the above output notice that each "type" of package is dealt with separately. Hence if the package "pkg2" had 10 versions installed, but five each were for "Linux" and "HP-UX" respectively - then 5 copies of each would be kept.

8.5 Automatic Repository Management

Although it is possible to schedule jobs to keep the content of a repository clean by deleting older packages it is sometimes not convenient. To help address this requirement it is possible to create a file in the repository directory which is used whenever the repository index is rebuilt. This file is called:

```
.repos_options
```

At present only two options are available to add to this file:

Option	Purpose
FORCE (0 1)	Force repository index rebuild or not. If not then packages with bad permissions might cause the index not to be built.
KEEP nn	Indicates the number of versions of each package to keep.

It should be noted that if a repository contains the same versions of a package, but for different operating systems and or architectures they are considered separate packages, and so the maximum number of versions to be kept for each variant.

9 Package Cleaning and Post-Boot Actions

9.1 When do Packages need Cleaning?

If a package installation or removal is interrupted [either the process dies, the process is killed or the machine crashes/reboots. For example consider the following aborted session:

```
Log :
Log : Installation of pigpkg starting - please wait.
Log : Package file : pigpkg+hpux+1.0.0.tp2
Log : Log file      : install+pigpkg+1182259359
Install bin/3300          ##### 33%
Error: Aborting after RC=130 from installation of pigpkg.
Error: [Use "tp2log --namespace test1 --show install+pigpkg+1182259359" for
details]
```

In this instance attempting to install the package again is not directly possible:

```
Log : Obtaining lock[write] on test1 ... Got it.
Error: Package is partially installed - please run cleanup.
```

This is more evident if you examine the namespace packages:

```
$ tp2 list --namespace test1
Package  Version  State      Installed
=====  =====  =====  =====
fred     1.1.0    Installed  14/06/2007
newfred  2.1.0    Installed  14/06/2007
pack2    1.1.0    Installed  18/06/2007
pigpkg   1.0.0    Installing
testpkg  1.0.0    Installed  14/06/2007
```

This is an example of a corrupt package that needs to be recovered. The same can also of course happen when a package is being removed.

9.2 How to Clean Packages

When a package is in either a "Removing" or "Installing" state it needs to be "cleaned". This will ensure that the namespace is taken to a sane state. This sane state might re-instate a package that is being removed, or remove a package that is partially installed.

This action is performed using the "tp2 clean" command which works on a complete namespace. For example:

```
$ tp2 clean --verbose test1
```

The verbose option is always useful - and this is one of the view commands in TP2 where a preview option is not available. The output generated indicates that example package shown above is being cleaned:

```
Log : Obtaining lock[write] on test1 ... Got it.
Log : Getting package details for namespace "test1"... Done [5 packages].
```

```
Log : Cleaning package "pigpkg" [state=Installing]
Log : Clean successfully [See "tp2log --namespace test1 --show
clean+pigpkg+6593"]
Log : Packages scanned      : 5
Log : Successfully cleaned  : 1
Log : Unsuccessfully cleaned : 0
```

Following this action notice that the package has been removed:

```
$ tp2 list --namespace test1
Package  Version  State      Installed
=====  =====  =====  =====
fred     1.1.0    Installed  14/06/2007
newfred  2.1.0    Installed  14/06/2007
pack2    1.1.0    Installed  18/06/2007
testpkg  1.0.0    Installed  14/06/2007
```

Things are different when a package is partially removed - it is shown as having a "removing" state:

```
Package  Version  State      Installed
=====  =====  =====  =====
fred     1.1.0    Installed  14/06/2007
newfred  2.1.0    Installed  14/06/2007
pack2    1.1.0    Installed  18/06/2007
pigpkg   1.0.0    Removing
testpkg  1.0.0    Installed  14/06/2007
```

In this instance after the package will be fully installing if it was interrupted during "staging" of the package delete. It instead the interruption was during the package "removal" stage then the clean action would actually complete the package removal.

For example:

```
$ tp2 clean --verbose testns2
Log : Obtaining lock[write] on testns2 ... Got it.
Log : Getting package details for namespace "testns2"... Done [6 packages].
Log : Cleaning package "pigpkg" [state=Removing]
Log : Clean successfully [See "tp2log --namespace testns2 --show
clean+pigpkg+1257250471"]
Log : Packages scanned      : 6
Log : Successfully cleaned  : 1
Log : Unsuccessfully cleaned : 0
```

After the clean the package is still installed:

```
$ tp2 list --namespace testns2 -format -signer pigpkg
Package  Version  State      Installed
=====  =====  =====  =====
pigpkg   1.0.0    Installed  30/10/2009
```

After package cleaning it is strongly recommended that the namespace be verified for potential problems:

```
$ tp2 verify --namespace testns2 --verbose pigpkg
```

TP2: Administrator's Guide

```
Log : Loading configuration of packages in "testns2" namespace... Done [6 packages]
Log : Checking package "pigpkg".
```

10 Repackaging

10.1 What is Repackaging

"Repackaging" is the term used when an existing package is being replaced [whether downgrading, upgrading or re-installing the same version], and a back-out to the current system configuration is required.

The obvious question is what is the difference between re-installation of the version that has just been over-written? Indeed there is no difference - *as long as none of the files installed have been manually changed, replaced or deleted.*

Hence repackaging dynamically generates a copy of what is currently installed for a package, rather than what was installed when the current version of the package was originally installed. The use of such a facility should become clear over the next few sections.

10.2 When is repackaging typically used

Use of repackaging is strongly encouraged in all production environments. Of course the additional work that is performed should be considered - it does lengthen installation times significantly. However the benefit of being able to "roll-back" to the previous state is massive for production environments and gives administrators a greater confidence when installing packages.

The process of generation and capping the amount of storage that such repackages use can be made automatic once the namespace is configuration correctly and so does not increase administrator workload in any way.

It is even possible to ensure that repackages are *always* performed even if not requested - as an added benefit for system administrators not wishing to worry about the number of command line options they must specify!

10.3 Examples

To see the use of repackaging consider the following example using a dummy package. Firstly consider the following "pack2" package:

```
$ tp2 list --namespace test1 --detail pack2
Name      : pack2
Version   : 1.1.0
Status    : Installed
Scripts   : Postinstall,Preinstall,Preremove
Altrepos  : WWW:myhost.net/tp2packages
Architecture : 9000_800
Dependencies : testpkg|1.0.0|99999999.999999.999999
Description : An example test package2
Incompatibles : oldfred|0.0.1|99999999.999999.999999
Files     :
bin/ls    : a75124da1ecb6221976e0674e6b477fb97877685 37
bin/cat   : fb1c5c8f9be77249a0717656956fa2156df75bb2 20
```

So the package has just two files, with the file "bin/cat" 20 bytes long. However that on disk the file has been manually modified:

```
$ ls -l /tmp/test1_root/bin/cat
-rwxr-xr-x  1 venture  users          51 Jun 26 08:24
/tmp/test1_root/bin/cat
```

- Of course the fact that the file has been altered would have been picked up if the user had run the "tp2 verify" utility on the namespace.

At some point later the administrator is asked to reinstall the package because a user has deleted a file. Hence the administrator runs the following command:

```
# tp2 install --namespace test1 --verbose --pkg pack2 --repos /tmp \
--loose --version 1.1.0 --reinstall
```

The administrator then verifies the configuration and finds the package is installed without problems:

```
$ tp2 verify --namespace test1
```

So he is happy and the user that deleted a file is happy problem fixed... But then another user complains - the "bin/cat" program does not appear to be working! If repackaging was not used then the cause of this problem would never be found - [since a verification of the package before re-installation was not performed]. But fortunately the administrator had the foresight to automatically enable repackaging and knows that the repackaged versions of programs for this namespace exist in a "/tmp/repackages" directory:

```
$ tp2 list --repos /tmp/repackages
Package  Version  Size      Description
=====  =====  =====
=====
fred     1.1.0     24576    Repackaged 1.1.0 during 1.1.0 install - An
example fred package
pack2    1.1.0     24576    Repackaged 1.1.0 during 1.1.0 install - An
example test package2
pigpkg   1.0.0     82632704 Repackaged 1.0.0 during 1.0.0 install - An
example bad package
testpkg  1.0.0     24576    Repackaged 1.0.0 during 1.0.0 install - An
example test package
```

Notice that the description indicates which version was installed at the time - very useful if a lot of packages change. Hence the administrator runs the following command:

```
$ tp2 install --preview --namespace test1 --verbose --pkg pack2 --repos
/tmp/repackages --loose --version 1.1.0 --reinstall
```

Checking the generated preview installation log shows the following:

```
Log : Arguments: --namespace=test1 --pkgfile
/var/adm/tp2/ns/test1/spool/pack2-repackaged+9000_800+1.1.0.tp2 --verbose
--preview --logfile /var/adm/tp2/ns/test1/log/install+pack2+1182844381
Log : Namespace security checks passed.
Log : Successfully spooled package contents to /tmp/test1_root/pkginstall.7710
Log : Existing package - implicit upgrade/downgrade will be performed.
Log : Overwriting /tmp/test1_root/bin/cat
[fb1c5c8f9be77249a0717656956fa2156df75bb2 ->
95ccb40397de63e3ecd907ebb3e07546626e06e4]
Log :
Log : Result: SUCCESS [ 0.39 secs User CPU , 0.15 secs System CPU ]
```

So the repackaged version shows that "bin/cat" was different. In such cases this is usually caused by files being manually changed in-place by-passing packaging mechanisms and hence undermining release control.

Recovery from this scenario is fortunately quite straightforward - it is a matter of installing the package again from the repackages repository:

```
$ tp2 install --namespace test1 --verbose --pkg pack2 \
--repos DIR:/tmp/repackages --loose --version 1.1.0 --reinstall
```

Now checking the offending program it has been restored to the same value prior to the initial re-installation - so all should be well!

```
$ ls -l /tmp/test1_root/bin/cat
-rwxr-xr-x 1 venture users 51 Jun 26 10:18
/tmp/test1_root/bin/cat
```

10.4 Where are the Packages?

If repackaging is allowed and used where do the repackaged versions of the software reside? By default the location of such packages are:

```
/var/adm/tp2/ns/namespace/repackages
```

However it is possible to change this by adding a line such as the following in the namespace configuration file:

```
REPACKAGE_DIR /tmp/repackages
```

By default repackages continue to consume disk space as and when required. Looking in a directory that is used for repackages each package can be seen as a simple file:

```
$ ls -al /tmp/repackages
total 624
drwxrwx--- 2 venture u 8192 Jun 26 08:52 .
drwxrwxrwt 161 root u 49152 Jun 26 10:40 ..
-rw-rw---- 1 venture u 927 Jun 26 08:52 .repos
-rw-rw---- 1 venture u 489 Jun 22 15:00 fred-repackaged+1.1.0.tp2
-rw-rw---- 1 venture u 692 Jun 26 08:52 pack2-repackaged+9000_800+1.1.0.tp2
```

```
-rw-rw---- 1 venture u 223614 Jun 22 14:36 pigpkg-repackaged+hpux+1.0.0.tp2
-rw-rw---- 1 venture u 655 Jun 26 08:17 testpkg-repackaged+1.0.0.tp2
```

The administrator can remove files manually and regenerate the index file using "tp2 repos" if they wish, however it is more usual to allow TP2 to self-manage this space using the following configuration file entry:

```
REPACKAGE_MAXSPACE 10
```

This defines the maximum amount of space the packages can occupy [in Megabytes]. Following a package installation session that makes of repackaging the directory will be examined and all older packages removed until the space occupied falls below this value.

10.5 Restoring a Re-packaging version

As shown in the example installation of a package that is actually a repackage is no different. The administrator may need to make use of the "--reinstall" or "--downgrade" flags and also specify a particular version using "--version". For example:

```
$ tp2 install --namespace test1 --verbose --pkg pack2 \
--repos DIR:/tmp/repackages --loose --version 1.1.0 --reinstall
```

There are however two minor differences when installing a package that is a repackage;

- If the package is being installed from the directory known for repackages for the namespace no repackage will be performed [since this would fail when a checksum error since the repackage and the package to install will be the same].
- The description used for the package indicates it is actually a repackaged version of the software.

```
$ tp2 list --namespace test1 --format "package,version,description"
Package  Version  Description
=====  =====
-----
fred     1.1.0    An example fred package
newfred  2.1.0    A newfred package
pack2   1.1.0    Repackaged 1.1.0 during 1.1.0 install - An example test
package2
pigpkg   1.0.0    An example bad package
testpkg  1.0.0    An example test package
stbe905a$ mci --msg "Support description column in sort" tp2list
```

11 Log File Management

11.1 What is logged?

TP2 understands the importance of tracking any changes that might occur in a namespace. Hence it aims to keep detailed logs whenever possible. At present the following activities currently generate logs:

- Package Installation [whether real or preview]
- Package Removal [again for both real and preview removals]
- Namespace Cleaning
- Post Boot Configuration

The location of log files can not currently be changed. As usual TP2 provides mechanisms for ensuring logs do not consume more disk space than necessary. There are several directives that can be added to a namespace configuration file to manage log space:

Option	Purpose
MAXLOGS	The maximum number of logs to keep, older logs will be removed. If not specified will default to 50.
MAXLOGSPACE	The maximum amount of space the logs can occupy (in MB). Older logs will be removed until under this threshold. If not specified will default to 2.
LOG_MAX_PREVIEW_AGE	By default preview logs are dealt with as normal logs. Using this parameter it is possible to ensure preview logs are deleted after a certain age to help preserve disk space. Defaults to "-1" meaning preview logs are not deleted any differently than normal logs.
LOG_COMPRESS	Set to 0 or 1. If "1" then log files are automatically compressed after they are generated. Defaults to 0 if not specified.
LOG_COMPRESS_TYPE	By default logs will be compressed with "compress" (if available). Use this to set it to the path of "compress", "gzip" or "bzip2" to compress logs using a different algorithm.

All directives are optional and can be combined in any way. Typically the "LOG_COMPRESS" is turned on and the amount of logs and space they occupy increased for production environments.

11.2 Examining Logs

Whenever an action occurs that results in a log being generated that log will exist as a separate file. Assuming you have permissions to access the logs you can view them for a namespace via the "tp2 list" command. For example:

```
$ tp2 list --namespacelogs test1
Action  Object  Date           Preview?  Result  Reference
=====  =====  =====
Install pack2    22/06/2007 10:38 No        Success  install+pack2+1182505117.gz
Install pack2    22/06/2007 10:40 No        Success  install+pack2+1182505219.gz
Install pack2    26/06/2007 08:53 Yes       Success  install+pack2+1182844381.gz
Install pack2    26/06/2007 10:18 No        Success  install+pack2+1182849527.gz
Remove  pack2    26/06/2007 16:33 No        Fail      remove+pack2+1182872009
Remove  pack2    26/06/2007 16:33 No        Success  remove+pack2+1182872015
```

The output will be the same even if the logs are compressed. The compression is transparent - logs are compressed when necessary and automatically uncompressed when their contents are needed.

Since there might be a large number of logs it is also possible to use the "--type" option to limit the output just to certain types of log entries. The values excepted by this option are "install", "remove", "boot" and "clean". For example:

```
$ tp2 list --namespacelogs test1 --type clean
Action  Object  Date           Preview?  Result  Reference
=====  =====  =====
Clean   pigpkg  22/06/2007 11:41 Unknown  Fail      clean+pigpkg+24666.gz
Clean   pigpkg  22/06/2007 11:42 Unknown  Fail      clean+pigpkg+1193.gz
Clean   pigpkg  22/06/2007 11:44 Unknown  Fail      clean+pigpkg+4203.gz
Clean   pigpkg  22/06/2007 11:50 Unknown  Fail      clean+pigpkg+15360.gz
```

It is possible to restrict the output to logs for a certain packages by specifying the names of the packages at the end of the command line. For example:

```
$ tp2 list --namespacelogs test1 testpkg
Action  Object  Date           Preview?  Result  Reference
=====  =====  =====
Install testpkg  22/06/2007 14:41 Yes       Success  install+testpkg+1182519683.gz
Install testpkg  26/06/2007 08:17 No        Success  install+testpkg+1182842275.gz
```

Of course it is also possible to combine the "--type" option with specification of the package name to further refine the output. The package names that are specified can include the "*" character to act as a wildcard if necessary. Finally the "--days" argument ensures only activities less than the specified number of days old is included:

```
$ tp2 list --namespacelogs test1 "*pkg" pack2 --type install --days 3
Action  Object  Date           Preview?  Result  Reference
=====  =====  =====
Install testpkg  26/06/2007 08:17 No        Success  install+testpkg+1182842275.gz
Install pack2    26/06/2007 08:40 No        Success  install+pack2+1182843647.gz
Install pack2    26/06/2007 08:51 Yes       Success  install+pack2+1182844317.gz
Install pack2    26/06/2007 08:52 No        Success  install+pack2+1182844371.gz
Install pack2    26/06/2007 08:53 Yes       Success  install+pack2+1182844381.gz
Install pack2    26/06/2007 10:18 No        Success  install+pack2+1182849527.gz
```

To display details of a particular log the "tp2 log" command is used. For example to display the contents of the log shown in the previous output in bold, the following command would be used:

```
$ tp2 log --namespace test1 --show install+pack2+1182844371.gz
```

Notice that the "reference" column is used to uniquely identify the log file in question. This can also be a pattern to show multiple logs. Hence to show all logs for installation of the "pack2" package the following command is possible:

```
$ tp2 log --namespace test1 --show install+pack2+*
```

11.3 Managing Log Space

As stated previously TP2 will automatically limit the amount of space occupied by logs. However this can further be reduced by compressing the logs with "Gzip". For example to compress a log to consume less space use the "--compress" argument with the reference of the log in question. Again patterns can be specified if required to compress multiple files.

For example to compress all files that are not currently compressed use a command such as:

```
$ tp2 log --namespace test1 --compress "*" --verbose
```

Any logs that are already compressed will of course be ignored.

12 Namespace and Package Security

12.1 Basic UNIX-level security considerations

The concept of namespaces is a key defining feature of TP2 and thus care must be taken to ensure that each namespace is secure. For example if "user A" owns namespace "Y", then "user B" should not be able to alter the contents of that particular namespace.

By default it is possible to view the contents of any namespace, whatever the user, but not make any changes to it. However it is possible to turn even this level of functionality off - allowing namespaces to be essentially "private".

When a new namespace is created it will be given a directory under "/var/adm/tp2/ns" for the namespace in question, for example:

-rw-r--r--	1	root	sys	107	Jul	14	14:20	config
drwxrwxrwt	2	user	sys	96	Nov	5	11:28	locks
drwxr-xr-x	2	user	sys	8192	Nov	3	07:44	log
drwxr-xr-x	2	user	sys	8192	Oct	30	09:21	meta
drwxr-xr-x	2	user	sys	8192	Nov	3	07:44	pkg
drwxr-sr-x	2	user	sys	96	Jul	14	14:21	pubkeys
drwxr-x---	2	user	sys	96	Jan	5	2007	repackage
drwxr-xr-x	2	user	sys	8192	Nov	3	07:44	spool

The entries "locks" and "pubkeys" are optional and might not exist. The existence and permissions on the "locks" directory control the security for a particular namespace. The following simple rules are followed when attempting to access a namespace:

- Before anything takes place work out what lock to get; "read-only" for a query operation; "read-write" for an update operation.
- If lock == "read-only" and the "locks" directory exists and is writable by the user then attempt to grant a read lock using the ".lock" file in that directory.
- If no suitable lock file location has been defined then attempt to generate a ".lock" file to lock in the "logs" directory - this only works for the user (or "root") that owns the namespace.

The logs directory should be set to permissions "1777" octal if it exists to ensure consistent locking behaviour. The directory can be added/removed as and when needed.

By default only "root" can perform many of the administration tasks for TP2 though this can be changed using the TP2 Daemon - see "TP2 Security and Auditing Daemon" on page 52 for details.

12.2 Package Signing Concepts

TP2 supports packages that are signed. Currently a package can only be signed by one individual (though this functionality is set to be extended in later versions). Signed packages have been introduced to allow users to validate that a package they wish to install is actually from a valid source.

The process of signing a package is very easy; the user that generates the package uses the "tp2 signing" command and then the package can be placed in a repository and installed as normal.

A signed package differs internally in that it contains two parts; the actual package and a signature. The signature of a package that is signed by a user will be unique to this package for this signer.

Any namespace can be configured to either ignore signatures; warn if packages do not use them; or fail if packages do not use them. When a signed package is installed the signature is compared to the details available for the user that signed the package, and only if deemed valid will the package actually be installed.

Of course, the key here is the strength of the signing mechanism. To perform this signing a user first creates a public/private key pair using the well known DSA algorithm. Assuming the private key is kept safe it means that the specified public key can only validate packages signed using the associated private key, and can not be theoretically spoofed.

Hence the typical scenario for using a signed package is initially to download and install the public key of the signer you wish to trust, and then install packages as normal - nothing complicated at all.

12.3 Generating a Public/Private Key Pair

If anyone wishes to sign a package they must define a public and private key pair. This is done using the "tp2 signing" tool.

```
$ tp2 signing --keygen
```

The above command may take a little time to run, but once complete it will have generated two files in the user's home directory:

```
-r----- 1 venture users 404 2009-06-30 22:46 /home/venture/.tp2-venture.private
-rw-r--r-- 1 venture users 439 2009-06-30 22:46 /home/venture/.tp2-venture.public
```

Notice that the files are named in such a way that "tp2 signing" can use them to sign packages - do not rename them. Also the permissions should not be changed - the "private" file must be readable only by the signer. Keeping the private key unreadable is critically important to security.

Once a the public key is available it can be made available to users. It is strongly recommended that once you sign packages you do not re-create the key later. If you do you might have users with different public keys to the private key used to sign the packages, leading to confusion.

12.4 Signing a Package

Once a package has been created it can be signed. When a package is signed it typically replaces on disk the file that was the original package. Only if that package is not writable will an alternative name be used.

You do not need to have created a package to sign it. This is an important consideration allowing good role separation between the developers creating the package whilst other "approved" users are then able to sign it after the appropriate quality checks.

To sign a package again the "tp2 signing" command is used, this time in the format of:

```
$ tp2 signing --sign --pkg /tmp/tp2+1.1.1.tp2 --verbose
```

With the "--verbose" functionality this output similar to the following:

```
Log : Loading public key [/home/venture/.tp2-venture.public].
Log : Loading private key [/home/venture/.tp2-venture.private].
Log : Reading package contents to memory and generating signature - please wait...
Log : Signing package of 178573 bytes.
Log : Package contents SHA1: 6114d6b1210ee2f68803f1923b3e8960c61fe8a3
```

By default if a package is already signed it cannot be signed again:

```
$ tp2 signing --sign --pkg /tmp/ss/tp2+1.1.4.tp2 --verbose
Log : Loading public key [/home/venture/.tp2-venture.public].
Log : Loading private key [/home/venture/.tp2-venture.private].
Error: Specified package is already signed.
```

To replace a signature add the "--replace" option to the command, for example:

```
$ tp2 signing --sign --pkg /tmp/ss/tp2+1.1.4.tp2 --verbose --replace
```

Obviously to sign packages you need write access to the directory where the package resides. If you are unable to overwrite the original package (due to permissions on the directory typically), then the software will generate the package prefixed by your username.

```
Log : Signed package is named '/tmp/ss/venture-tp2+1.1.4.tp2' [could not
overwrite '/tmp/ss/tp2+1.1.4.tp2'].
```

```
Error: The namespace 'testns' has been configured to only accept
Error: packages that are signed by approved personnel. The package
Error: specified has no DSA signature.
Error: Please get one of the following to sign the package:
Error: * simon.edwards@mynet.com
Error: * fred.blogs@ournet.com
```

```
Error: The namespace 'testns' has been configured to only accept
Error: packages that are signed by approved personnel. The package
Error: specified has no DSA signature.
Error: NO VALID PUBLIC DEFINED FOR NAMESPACE YET.
```

13 TP2 Security and Auditing Daemon

13.1 Purpose of Daemon

By default most actions that concern the create and management of namespaces take place as "root". Once a namespace is created the designated owner can manage many aspects related to the packages that might be present in that namespace, but not how some aspects of the namespace work.

For example only "root" can alter the configuration file for the namespace. This means that as a normal user you cannot change options regarding packaging signing, or repackaging, or how much space logs to consume for example.

Also by default when a package is installed, replaced, removed or changed the status is logged only with the namespace in question – there is no central log of all changes that occur across all namespaces.

Also typically if a package is too large to be installed it might fail to install, even though the file system might be grown the user would not have permissions to do so.

All of the above scenarios can be overcome by using the "TP2 Daemon". This is an optional component that can be enabled and typically runs automatically as part of the machine boot sequence.

Of course security is a key aspect of the daemon – the system administrator can configure which users have access to the facilities it offer, thus controlling access to some of the more unique features; such as automatic file system expansion.

If a system has lots of namespaces managed by lots of different users it is often the best way of allowing certain functions can happen without continual intervention by the system administrator; who most likely has lots of other things that must be done on their mind!

13.2 General Configuration and Start-up

Typically the TP2 daemon is run when the server is started by using the "Run Control" mechanism based on the operating system and/or distribution. For example most UNIX's might use "/etc/init.d" for processing, some Linux distributions instead use "Upstart", whilst Solaris uses SVC.

No matter the mechanism for actually starting the daemon it will a configuration file to work. At present the configuration file is limited. Typically it is expected to exist in one of the following locations based on the standard start-up script:

```
/usr/local/tp2
/opt/tp2
/opt/tp2/etc
/etc/rc.config.d
```

This file defines whether to start the daemon, and typically might have the following contents:

```
TP2_DAEMON=1
TP2_DAEMON_CONFIG="/var/adm/tp2/tp2daemon.xml"
```

The above indicates that the daemon should be run and the configuration file to use is "/var/adm/tp2/tp2daemon.xml".

The contents of the configuration file, as the name suggests, are XML based, and typically an example might be:

```
<?xml version="1.0" standalone="yes"?>
<tp2daemon>
  <port>22777</port>
  <allow_namespace_users>user1,user2</allow_namespace_users>
  <allow_namespace_changes>repackaing,storage,logging</allow_namespace_changes>
  <audit_log>/var/adm/tp2/audit.log</audit_log>
</tp2daemon>
```

It should be noted that the "allow_namespace_users" is not currently used and the contents of this are ignored. All other options are used and now described:

Element	Purpose
port	The TCP port that the daemon listens on. For purposes of security only connections from "localhost" are accepted.
allow_namespace_changes	This defines the type of changes that the TP2 daemon should allow. Unfortunately this covers all namespaces, and can not map to individual ones as yet. At present this entry is parsed, but not actually enforced.
audit_log	The name of the audit log to write records to. This is not a large file but should be secure. It is written to as "root" and only "root" should have write-access to the contents.

13.2.1 Entries available for "allow_namespace_changes"

Although the current release does not enforce this functionality, it does parse it so the values must come from the items in the following table. It is a comma-separated list of values. Further values are likely to be added in the future, as well as namespace-specific configuration options.

Attribute	Purpose
repackaging	Allow repackaging in directories defined in the namespace that are not directly writable by the user owning the namespace where the repackaging is taking place.
storage	Allow TP2 daemon to grow volumes/file systems if allowed by a namespace that is too small to accommodate a package being installed.
logging	If present will allow centralised-logging of TP2 software changes across the server.
repositorymgt	Allow users to manage repositories they do not directly have access to (security of each repository defined by a ".tp2repos-security" file in repository root directory).

13.3 User Configuration Steps

Once the daemon is running it will only accept requests from users that are defined via files in the following directory:

```
/var/adm/tp2/security
```

In this directory each user has a file named as their user account. This file contains a string used as a two-way encryption key to communicate with the tp2daemon process. The contents can be any character string less than 30 characters.

The user has the same value stored in the following file in the user's home directory:

```
.tp2key
```

The permissions on this file should be octal 400. If they are not the key is not acceptable and will not be used by the utilities. Of course you will need "root" to install the key in the "/var/adm/tp2/security" directory.

Once a new key has been added to the above directory the running daemon must be sent a signal to ensure it re-reads the security directory details to pick up the new users, for example:

```
# ps -ef | grep tp2daemon | grep -v grep
  root 23384      1  0  Oct  6  ?           0:00 /usr/bin/perl -w ./tp2daemon -C
/var/adm/tp2/tp2daemon.xml
# kill -HUP 23384
```

Of course when removing users simply remove their usernames from the "/var/adm/tp2/security" directory and send the daemon a signal again to re-read user credentials.

13.4 Available User-available Functionality

Once a user has been defined as a user known to the tp2daemon they gain further functionality that previously only "root" was able to perform, including:

- Namespace creation, removal and modification
- File system expansion (when supported by underlying storage)
- Allow packages to be signed/built in repositories user does not have direct access to (to be implemented).

Once a user has been configured with access the following commands can be used (as explained previously in this guide):

```
tp2 make_ns ...
```

```
tp2 remove_ns ...
```

```
tp2 mod_ns ...
```

```
tp2 install ...
```

Notice that the tp2 installation can only take place in name spaces you own, but if the namespace is configured to support automatic file system expansion and the storage architecture supports it the tp2 daemon process will grow file systems as necessary.

14 General Namespace Administration Topics

14.1 Namespace Verification

One utility that should be used on a regular basis is the "tp2 verify" routine. This takes a particular namespace and validates the status of all packages in that namespace. This utility is very straightforward to run:

```
# tp2 verify --verbose --namespace test
```

When run it performs the following checks for every file and directory for every package in the specified namespace:

- The size of the file is correct [if not considered a "volatile" or a "configuration" file]
- The owner, group and permissions are correct.
- The SHA1 checksum for the file contents are current [apart from "volatile" or "configuration" files again].

The verbose option also indicates when a particular file is owned by more than one package. A sample of output might look like the following:

```
Log : Loading configuration of packages in "test1" namespace... Done [5 packages]
Log : Checking package "fred".
Error: Incorrect file permissions: /tmp/test1_root/bin/mycp [is 444, should be 755]
Log : Checking package "newfred".
Log : File overwritten by fred: /tmp/test1_root/bin/mycp
Error: Incorrect file permissions: /tmp/test1_root/bin/mycp [is 444, should be 755]
Error: Incorrect file group      : /tmp/test1_root/bin/stuff [is sasadmin, should be users]
Log : Checking package "package3".
Error: Incorrect file permissions: /tmp/test1_root/bin/mikky [is 640, should be 755]
Log : Checking package "pigpkg".
Log : Checking package "testpkg".
```

If you wish to check for one or more packages place the names of the packages at the end of the command line, for example:

```
$ tp2 verify --namespace test1 newfred fred
Error: Incorrect file permissions: /tmp/test1_root/bin/mycp [is 444, should be 755]
Error: Incorrect file group      : /tmp/test1_root/bin/stuff [is sasadmin, should be users]
Error: Incorrect file permissions: /tmp/test1_root/bin/mycp [is 444, should be 755]
```

As usual pattern matching can be used as well - so the above command could also be achieved using the following instead:

```
$ tp2 verify --namespace test1 "**fred**"
```

14.2 Namespace Permissions Fixing

Of course being able to report on problems is only part of the issue; the real requirement is actually being able to fix the scripts. To help with this the "tp2 verify" option also supports a "--fixscript" argument which actually generates a shell script that can be run to resolve problems.

For example consider the following command:

```
$ tp2 verify --namespace test1 "*fred*" --fixscript=/tmp/fixes
```

This generates a file called "/tmp/fixes" with the following contents:

```
chmod 755 /tmp/test1_root/bin/mycp
chmod 755 /tmp/test1_root/bin/mycp
chgrp users /tmp/test1_root/bin/stuff
```

Hence to fix the problems just run the script:

```
$ ksh /tmp/fixes
```

Of course once this has been fixed any problems with permissions. The scripts can sometimes look different if the utility needs to attempt to fix an ownership problem for a file and the user is not running as root. For example consider the following problems:

```
Error: Incorrect file owner      : /tmp/test1_root/bin/fff [is root, should be venture]
```

In this case the fix script will look like the following for this problem (formatted nicely for inclusion here):

```
# Trickery required since not owner any more.
mv -f /tmp/test1_root/bin/fff /tmp/test1_root/bin/fff.orig.$$ &&
cp /tmp/test1_root/bin/fff.orig.$$ /tmp/test1_root/bin/fff &&
chgrp users /tmp/test1_root/bin/fff &&
chmod 755 /tmp/test1_root/bin/fff &&
rm -f /tmp/test1_root/bin/fff.orig.$$
```

This type of change will only work if the user in question was write access to the directories where such files reside of course.

14.3 First Boot Scripts

Occasionally when examining software installation logs the following line might be seen:

```
Log : First boot script installed successfully.
```

What this does is to install a flag to indicate when the machine is next rebooted [and the "tp2 boot" command run as "root"], then the "firstboot" script should be executed for the package in question.

TP2: Administrator's Guide

When the machine boots the script in question will be run to finalise the package installation if necessary [such as removing all configuration files etc]. The "firstboot" scripts get run as the owner of the namespace where they are found - and not by "root".

Of course you may wish to attempt to run these manually at any time using the "tp2 boot" command, for example:

```
$ tp2 boot --verbose test1
```

The above command would perform post-boot processing on the "test1" namespace - including running any "firstboot" scripts.

```
Log : Post-boot cleaning/configuration namespace "test1"... Done.
Log : Namespace successfully cleaned.
Log : [Use "tp2log --namespace test1 --show boot+postboot+1183455099" for details]
Log :
Log : Namespaces Successfully cleaned : 1
Log : Namespaces Unsuccessfully cleaned : 0
Log : Namespaces unchanged : 0
```

In the above example it indicates that some cleaning was performed on the namespace, so viewing the log is recommended:

```
$ tp2 log --namespace test1 --show boot+postboot+1183455099
Log : Changing effective user to venture.
Log : Result: SUCCESS [ 0.34 secs User CPU , 0.06 secs System CPU ]
Log : Executed 1 first boot script.
Log : Deleted 0 files, 0 left in place.
```

In this case it indicates that no package cleaning or configuration was required.

Appendix A: Namespace Configuration File Options

This appendix includes details of all currently allowable entries in the configuration file for a namespace.

Attribute	Mandatory?	Example Value	Description
ROOT	Yes	<code>/tmp/test1_root</code>	The top level directory under which files for the package will be installed.
OWNER	Yes	<code>venture</code>	The name of the owner of the namespace. Only "root" or this user is able to install packages here.
MAXLOGS	No	<code>500</code>	The number of log files to keep for the namespace. Minimum value is 50.
MAXLOGSPACE	No	<code>10</code>	The number of MB of disk space for logs for this namespace. If not specified it defaults to 2 [minimum supported value].
LOGCOMPRESS	No	<code>1</code>	A binary value. If set to "1" log files are automatically compressed as soon as they have been completed.
LOG_COMPRESS_TYPE	No	<code>/usr/bin/gzip</code>	The program to use to compress logs. Recommended to indicate the full path, though might not be necessary.
TMPDIR	No	<code>/var/adm/tp2/ns/test1/spool</code>	The name of the directory to copy the packages to before they are unpacked. If not set will default to "spool" under the namespace meta data directory.
MAXSPOOLSPACE	No	<code>50</code>	The amount of space in the spool directory for downloaded packages. If not specified it defaults to "0" - meaning all spooled contents are removed as soon as the current use of them is complete.
REPACKAGE_ALWAYS	No	<code>1</code>	No value necessary. If this line is present that a repackaging of any currently installed package being changed will always occur.
REPACKAGE_DIR	No	<code>/var/tp2/repackaged</code>	The directory to store the repackaged versions to. If not specified they will go to <code>/var/adm/tp2/ns/repackage</code> directory.
REPACKAGE_DISALLOW	No		No value necessary. If this option is specified it will never be possible to use the <code>--repackage</code> option. If both <code>REPACKAGE_ALWAYS</code> and <code>REPACKAGE_DISALLOW</code> are both present then <code>REPACKAGE_ALWAYS</code> will take precedence.

Attribute	Mandatory?	Example Value	Description
REPACKAGE_MAXSPACE	No	100	The maximum amount of disk space the repackaged versions of programs should occupy. By default when repackaging is performed no limits on repackaged versioning information is imposed.
REPOS	No	HTTP:host/packages /tmp	A space separated list of repositories to fall back to if the TP2_PATH variable is not set and the current directory can not help.
UNSIGNED	No	WARN	Indicates what the namespace should do when unsigned packages are installed. Three values are supported: <ul style="list-style-type: none"> • ALLOW – unsigned packages are allowed. • WARN – warn when installing unsigned packages. • NEVER – previews allowed of unsigned packages, but not actual installs.
AUTOFSGROW	No	1	This is a Boolean (set to "0" or "1"). The default is "0". When set to "1" installation of packages into a namespace may attempt to dynamically increase file systems sizes if the packages in question do not appear to fit.
AUTOFSUNIT	No	4	The size in MB that a file system is grown back. This defaults to "4". The actual increase might not be a multiple of this unit depending on the attributes of the underlying storage.
TRIGGER_POST_INSTALL	No		To be implemented.
TRIGGER_POST_REMOVE	No		To be implemented.

Appendix B: Automated File System Support Configurations

This particularly useful feature is available across a variety of operating systems. The functionality is limited to certain configurations, though this could be easily extended when suitable test environments are available. This functionality is taken from TrueCL - a high availability cluster toolset written by the author of TP2.

Supported Volume Managers

Volume Managers	Operating Systems	Comments
Veritas Volume Manager (VxVM)	Solaris, Linux, HP-UX, AIX ¹	<ol style="list-style-type: none"> 1. Uses basic functionality that is available across versions 3,4 and 5 of software. 2. The smallest unit of growth supported is quite small, though a value of 32Mb or larger is recommended.
Logical Volume Manager (LVM)	HP-UX, Linux	<ol style="list-style-type: none"> 1. Storage is allocated in set-size logical partitions, and so if the unit size allocated is not a multiple of this size it will be rounded up when necessary.
AIX Volume Manager (LVM)	AIX	<ol style="list-style-type: none"> 1. Same as with HP-UX/Linux LVM - set sized allocation, with automated rounded up when necessary.
Solstice Disk Suite (SOLVM)	Solaris	<ol style="list-style-type: none"> 1. Supported is limited to configurations that use logical partitions to store devices. Since this is the most flexible way of managing storage with Solstice this is not considered a significant limitation.
ZFS	Solaris, Linux ²	<ol style="list-style-type: none"> 1. ZFS supports normal and thinly provisioned storage based on storage pools. TP2 supports growth of either type of volume/file system.

¹ VxVM is available for AIX though this has not been tested.

² ZFS is available to Linux as a FUSE file system, though this has not been tested.

Supported File Systems

File System	Operating Systems	Comments
Veritas File System (VxFS)	Solaris, Linux, HP-UX, AIX ³	1. This is typically run on top of VxVM Volume Manager; though TP2 does not need this to be the case.
JFS	AIX, Linux	2. Works in an identical manner on both platforms from the point of view of TP2.
XFS	Linux	
ext2 / ext3	Linux	1. Both are treated identically and TP2 will work with either.
reiserfs3	Linux	
ZFS	Solaris, Linux ⁴	

Please note the following other file systems have not yet been tested:

- JFS2 - The latest version of this AIX file system is not significantly different from the command interface and so support can be added quickly if deemed necessary.
- BTRFS - The Oracle-sponsored next generated log-structured copy-on-write file system should be supported before end of 2009.
- ext4 - The same commands as ext2/ext3 are meant to be able to resize ext4 file systems too; though this has yet to be tested. If true support for this file system will be added quickly.

³ VxFS is available for AIX though this has not been tested.

⁴ ZFS file system functionality is available via FUSE, but has not yet been tested.