Distribution Constructor Design Specification

Table of Contents

Distribution Constructor Design Specification....1
1Overview....5
2Problem Areas....5
3Functionality....5
4Definition of Terms....5
5Overall Architecture....6
6Programming Language....8
7Distribution Constructor Application....8
7.1distro_const(1M) command....8
7.1.1Syntax:....9
7.1.2Descriptions....9
7.1.2.1distro_const build <manifest>....9
7.1.2.2-l option: distro_const build [-l] <manifest> 9
7.1.2.3-p option: distro_const build [-p <step>] <manifest>....9
7.1.2.4-r option: distro_const build [-r <step>] <manifest>....9
7.1.2.5-R option: distro_const build [-R] <manifest>....9
7.1.2.6-h option: distro_const [-h]....9
7.2Build Area....9
8Checkpointing....10
8.1Checkpointing Implementation 10
8.2Function interfaces....11
8.2.1DC_checkpoint_setup(cp, manifest_server_obj)....11
8.2.2step_setup(self, message, name, zfs_dataset)....11
8.2.3DC_delete_checkpoint(cp, step)....11
8.2.4DC_execute_checkpoint(cp, var)....12
8.2.5DC_verify_resume_step(cp, num)....12
8.2.6DC_remove_state_file(step)....13
8.2.7DC_verify_resume_state(cp)....13
8.3Data structures....13
8.3.1class step: 13
8.3.2class checkpoints:....13
9Manifest....14
9.1Format....14
9.1.1Style....14
9.1.2Design Considerations....15
9.2Content....15
9.2.1General....15
9.2.2Image Parameters....16
10Finalizer Scripts....17
11Execution Engine....18
11.1Programming Interfaces....18
11.1.1__init__(first_args_list=None)....18
11.1.2register(module, arglist=())....19

11.1.3change_exec_params(output=None, error=None, stop_on_err=None, logger_name=None) 19
11.1.4execute()....19
11.1.5get_exception()....20
12Manifest Parsing Module....20
12.1Parser Type....20
12.2Schema Type....20
12.3DOM Tree Access Abstraction Layer....21
12.3.1TreeAcc Class....21
12.3.1.1__init__(xml_file)....22
12.3.1.2find_node(path, starting_ta_node)....22
12.3.1.3replace_value(path, new_value, starting_ta_node)....22
12.3.1.4add_node(path, value, type, starting_ta_node, is_unique)....22
12.3.1.5save_tree(out_file)....22
12.3.1.6get_tree_walker()....23
12.3.1.7walk_tree(walker)....23
12.3.2TreeAccNode Class....23
12.3.2.1__init_(name, type, value, attr_dict, element_node)....23
12.3.2.2get_name()....23
12.3.2.3get_path()....23
12.3.2.4get_value()....23
12.3.2.5get_attr_dict()....23
12.3.2.6get_element_node()....23
12.3.2.7is_leaf()....23
12.3.2.8is_attr()....24
12.3.2.9is_element()....24
12.4Manifest Pre-Processing....24
12.4.1Defaults/Validation Manifest....24
12.4.1.1Helper Methods....24
12.4.1.2Defval Manifest vs DC Manifest....24
12.4.2Pre-Processing Sequence....25
12.4.3Defaults Processing....25
12.4.3.1Value....25
12.4.3.2from attribute....25
12.4.3.3path attribute....25
12.4.3.4type attribute....25
12.4.3.5missing_parent attribute....25
12.4.3.6skip_if_no_exist attribute....26
12.4.4Validation Processing....26
12.4.4.1Validate Path Processing....26
12.4.4.2Validate Group Processing....26
12.4.4.3Validate Exclude Processing....26
12.5Manifest Server and Reader Interfaces....27
12.5.1ManifestServ class....27
12.5.1.1__init__(manifest, valfile_base=None, out_manifest=None, verbose=False, keep_temp_files=False)....27
12.5.1.2get_values(request, is_key=False)....28

12.5.1.3start_socket_server(debug=False)....29
12.5.1.4stop_socket_server()....29
12.5.1.5get_sockname()....29
12.5.2ManifestRead class....29
12.5.2.1__init__(sock_name)....29
12.5.2.2get_values(request, is_key=False)....29
12.5.2.3print_values(request_list, are_keys=False, force_req_print=False)....30
12.5.2.4set_debug(on_off)....30
12.5.3ManifestServ commandline interface....30
12.5.4ManifestRead commandline interface....31
13Target Instantiation Module....31
13.1ti_create_target()....31
13.1.1function declaration....32
13.1.2Name-value pairs describing Target....32
13.2ti_zfs_list()....34
13.2.1function declaration....34
13.3ti_release_target()....34
13.3.1function declaration....34
14Logging....34
14.1exec_cmd_outputs_to_log(cmd, log, stdout_log_level=None, stderr_log_level=None)....35
15Finalizer Script Samples....35
15.1Populate package image area....35
15.2Root archive construction scripts....35
15.2.1Bootroot initialization script....35
15.2.2Bootroot archive script....36
16Transfer Module Modifications....36
16.1TM_perform_transfer()....37
16.1.1.1data structures....37
16.2Name – value Pairs describing Target....38
16.3TM_abort_transfer()....40
17References....40

1 Overview

The Distribution Constructor(DC) is a tool for building and customizing OpenSolaris installation images. Users specify the processing needed to create an image containing the desired content via a configuration file. DC also provides checkpointing and logging features to allow for rapid development and testing.

The Distribution Constructor tool will be used by Solaris Release Engineering, other OpenSolaris projects, such as the Slim Install project and the xVM project, OpenSolaris developers and OpenSolaris users who want to generate various types of distributions.

2 Problem Areas

Installable Solaris CD/DVD images have always been built by the Solaris Release Engineering(RE) organization. Solaris RE has an internally developed set of tools that they use for building these images. This set of tools is heavily dependent on Sun's internal system architecture, and they are not particularly well documented or easy to use. In addition, the RE tools require that all packages from Solaris be included in the image at all times. Unless one has access to these tools and knows how to modify them, there's no easy way to build an image with just a subset of the Solaris packages.

Many Solaris customers want to create their own custom deployment images. This is a feature that's commonly available in many other platforms (both Unix and Linux). For example:

- Fedora - Fedora Revisor (http://revisor.fedoraunity.org/)
- Ubuntu - Ubuntu Reconstructor (http://www.reconstructor.org)
- Suse - SUSE studio (http://susestudio.com/)
- HP-UX - HP-UX Ignite (http://docs.hp.com/en/IUX/)
- AIX - mksysb/mkcd

Solaris does not provide this feature. As a result, unofficial blueprint documents were written by field or marketing personnel to meet their customers' requirements. These documents are posted on sun.com without any product support. OpenSolaris needs to provide this functionality to be competitive.

3 Functionality

The Distribution Constructor (DC) is expected to eventually provide the following functionality. The implementation of items labeled “future” will not be discussed in this design specification.

- Users can select content and configuration of the resulting image a configuration file.
- Users can select content and configuration of the resulting image via GUI or CLI, then, save their selection into a configuration file. (future)
- DC will provide orderly execution of user provided scripts for creating the image.
- DC will support checkpointing during image creation. Users will be allow to list available checkpoints, stop, resume or re-execute a given checkpoint.
- Output and errors from the image construction process will be logged.
- DC will provide a set of finalizer scripts that are common for creating an image.

4 Definition of Terms

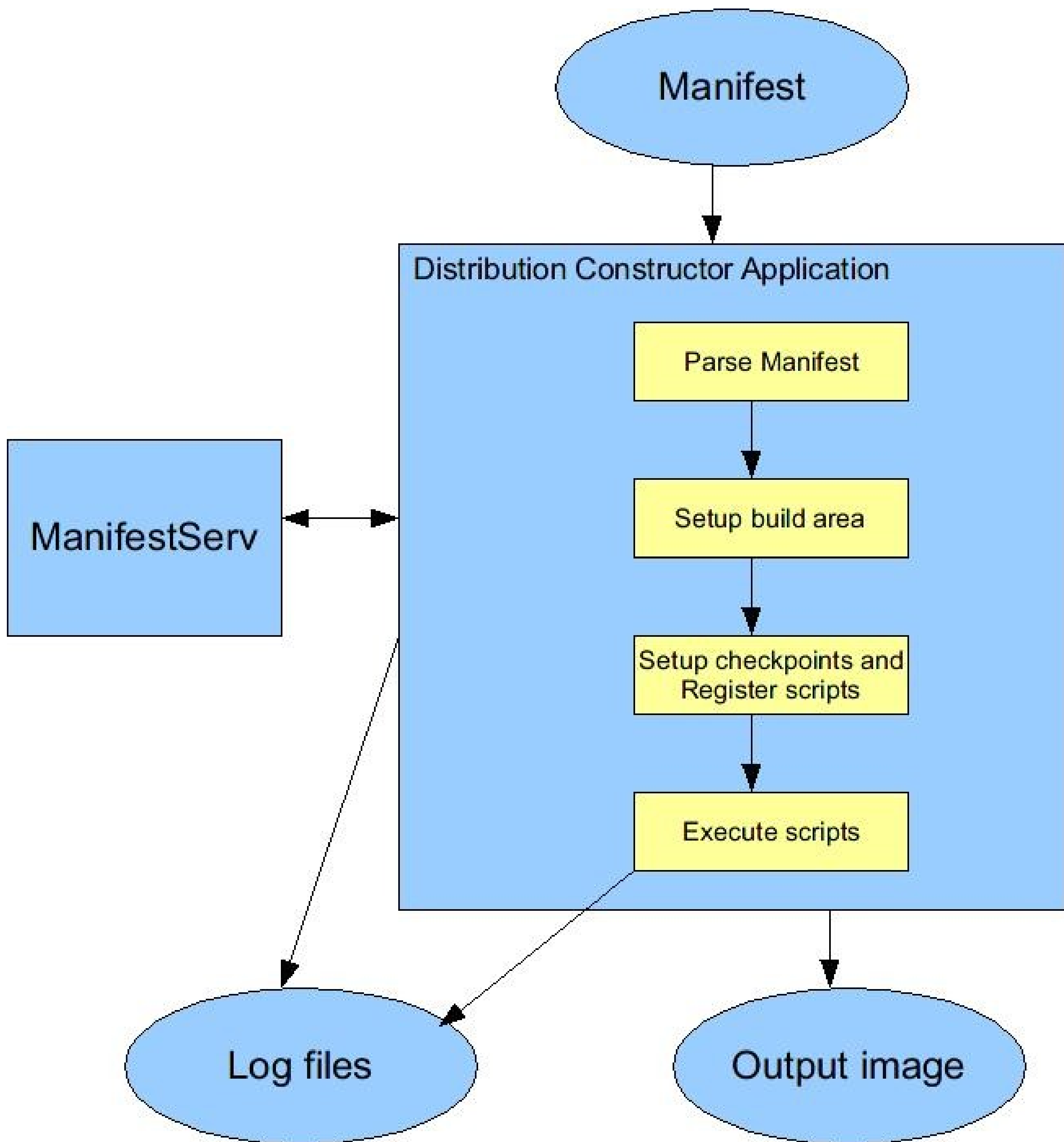
- Checkpoint: A snapshot in the image construction process.
- Manifest: A file containing all needed information for constructing the image.
- Finalizer scripts: scripts for creating the image.

5 Overall Architecture

The Distribution Constructor(DC) application drives the image construction process. It setups up the work area, parses the manifest and makes values from the manifest available to all finalizer scripts. It also implements the checkpointing feature. The DC, however, does not contain any logic on how images are created. It's up to the user to define and implement the image construction algorithms. The code to create the image is supplied as one or more finalizer scripts. The DC will execute those finalizer scripts in the order in which they are specified.

Some of the steps for creating images are common for most image types. For example, installing a specified set of packages from one or more IPS repositories, creating and configuring the root archive with a set of files, and creating ISO and USB image outputs. The DC project will provide a set of scripts for these common operations, which users can use directly for building images.

The following diagram shows how the DC works.



This is a brief summary of the functionality of each of the components. Interface and implementation details about each components can be found in subsequent chapters of this document.

Manifest

- This file contains all information needed to create the image.
- The file is in XML, so, it can be edited by hand.

- Detail information on manifest is at section 9

Manifest Parsing Module

- Validates and parses content of the manifest file.
- Creates a structure storing all information contained in the manifest file.
- Starts a server process making all manifest information available to finalizer scripts.
- Detail information on the manifest parser is at section 12

Setup build area(Target Instantiation Module)

- Prepares the work area for constructing the image. The work area could be just a regular directory or a ZFS file system.
- Detail information on the Target Instantiation Module is at section 13
- Detail information on layout of the build area is at section 7.2

Setup Checkpoints and register scripts

- Setup each finalizer script as a checkpoint, and register the scripts into the execution engine.
- Detail information on checkpointing is at section 8
- Detail information on finalizer scripts is at section 10

Execution Engine

- Calls modules which setup and customize the work area, and which generate distribution media or media images.
- The list and order of modules the finalizer module calls are specified in the manifest.
- Detail information on the execution engine is at section 19

Logging

- All outputs and errors from the Distribution Constructor application and all the finalizer scripts it executes are logged.
- Python logging is used.
- Detail information on logging is at section 14

6 Programming Language

The Distribution Constructor application as well as the different modules will be implemented in Python. The finalizer module will accept programs in C, shell scripts and Python. A detailed analysis of why Python is chosen is available here:
http://www.opensolaris.org/os/project/caiman/Constructor/design_notes/Language_Notes/

7 Distribution Constructor Application

The Distribution Constructor Application is the engine that drives the image construction process. The DC Application will be responsible for parsing the commandline arguments, calling the manifest parsing module, setting up the work area via the target instantiation module, initiating the finalizer module, and finally calling the finalizer scripts. Error handling/messaging, logging and checkpointing will be done at appropriate intervals between the modules.

7.1 distro_const(1M) command

/usr/bin/distro_const is the utility for creating distribution images and media

7.1.1 Syntax:

distro_const build [-R] [-r <step>] [-p <step>] [-l] <manifest>

7.1.2 Descriptions

7.1.2.1 distro_const build <manifest>

Builds the image from start to finish without pausing.

7.1.2.2 -l option: distro_const build [-l] <manifest>

Lists valid steps at which you can choose to pause or resume building an image. The step names displayed by this command are used as inputs to other checkpointing command options.

7.1.2.3 -p option: distro_const build [-p <step>] <manifest>

The build will from the start of the build process, and pause at the designated step. When combined with the -r option, it will resume execution at the designated step, and pause at the designated step.

7.1.2.4 -r option: distro_const build [-r <step>] <manifest>

The build will resume execution from the designated step. The specified step must be either the state at which the previous build stopped executing successfully or earlier. A state later than this is not valid. When combined with the -p option, it will resume execution at the designated step, and pause at the designated step.

7.1.2.5 -R option: distro_const build [-R] <manifest>

The will will resume from the last successfully executed step.

7.1.2.6 -h option: distro_const [-h]

Displays command usage. Command usage will also be displayed when any invalid option is specified.

7.2 Build Area

The build area is where the Distribution Constructor application do all it's work for creating an image. The location for the build area is specified by the user in the manifest. Build areas should be unique for each image. At the start of image construction processing, all content of the specified build area will be removed.

Build areas can reside in ZFS or UFS file system. If a build area resides in a UFS file system, checkpointing can not be used. To do it's work, the DC builds the following structure in the build area. <build_area>/build_data is a single ZFS dataset. <build_area>/media and <build_area>/logs are their own separate data set.

Directory	Comment
<build_area>/build_data/pkg_image	This is the package image area. All content from this directory is used for creating the final image.
<build_area>/build_data/bootroot	This is the build area for the root archive. The content of this directory will be archived into the root archive that will be placed in the image.
<build_area>/build_data/tmp	Temporary directory that can be used by the DC application or the finalizer scripts.
<build_area>/media	Output images
<build_area>/logs	Stores the log files for the build.

8 Checkpointing

In order to enable the user to perform multiple runs of the Distribution Constructor in a timely manner, DC provides checkpointing, which is the ability to stop and restart at any point of the image construction process. As discussed in section 7.1, the distro_const(1M) command provides multiple options for running a build using checkpoints.

The DC utilize the ZFS snapshot feature in the checkpointing implementation. ZFS snapshots of the work area are taken between each finalizer scripts so the state of the build can be saved. The user will then be able to restart the build at the checkpoint of their desire. If the work area specified in the manifest is a ZFS dataset, checkpointing is enabled automatically. If the user does not want to enable checkpointing when using a ZFS dataset as their work area, they can specify the option in the manifest. If the work area not on a ZFS filesystem or the manifest specifies a build area that isn't a mountpoint that directly corresponds to a zfs dataset, checkpointing will not be performed. If the user tries to use checkpointing, a message will be displayed explaining they need to be running zfs to use this feature.

When a checkpoint is taken, a copy of the manifest used when the checkpoint is taken is saved. All copies of saved manifests are stored as .step_<checkpoint_name> in the specified <build_area> directory. Inconsistencies between the manifest values used in the current run and the manifest values saved for a given state will be checked for and flagged as a warning. All .step_<checkpoint_name> files from the corresponding number on will be removed after verification has been performed when a new build is started. All corresponding zfs snapshots will also be removed at this time. If the checkpoint has a name, it will also be given a number internally to track where in the build sequence it occurs.

8.1 Checkpointing Implementation

The following is a high level description on the different functions that are called to setup and perform checkpointing. Detail descriptions of the functions are in the next section.

Initialization of the checkpointing code is done in the DC_checkpoint_setup() function. This code will check to see if checkpointing is available. If so, initialization for each step (or checkpoint) is performed via step_setup() function. The step_setup() function. will set up each step's basic information: name, message to print, state_file location, and the zfs snapshots to be taken. When the command line is parsed, the resume and pause steps are determined if applicable. Then DC_delete_checkpoint is called to remove any checkpoints that occur at or after the resume step. If no resume step is specified, all checkpoints are deleted. Throughout the build code, DC_execute_ checkpoint calls are performed at each place a checkpoint is desired. They must correlate with the above mentioned step_setup calls. When DC_execute_checkpoint is called, a check is made to determine if the following build code should be executed or not. If the current step is equal to or later than the resume step, a checkpoint is executed (zfs snapshot and .step file created) and the build code is performed. If the current step is before the resume step, the build code is not run. If the current step is earlier than the pause step, the checkpoint is created and the build code is run and then the build is exited. The case of no resume or pause step being specified is equivalent to the current step always being later than the resume step.

Checkpoints are performed between each finalizer script if checkpointing is available. The pause and resume steps are checked to see if the finalizer script should be run and if we want to rollback or checkpoint. The rollback and checkpointing is performed via finalizer scripts inserted in the queue before the specified finalizer script from the manifest.

8.2 Function interfaces

The following functions are in the DC_checkpoint.py file.

8.2.1 DC_checkpoint_setup(cp, manifest_server_obj)

Sets up and defines all checkpoints. This initialization will only occur if checkpointing is not disabled in the manifest file, and if the designated build area is a zfs dataset/zfs filesystem. All checkpoints must have an entry (in chronological order) here. For each step, create an entry in chronological order by calling the step_setup() function. In the DC_execute_checkpoint() function where the checkpoint to actually take place, the name specified in this function MUST be the same as the name used in the DC_execute_checkpoint call.

Inputs:
cp: the object containing all checkpoint related information
manifest_server_obj: handle to access the manifest server

Return code:
-1: failure
0: success

8.2.2 step_setup(self, message, name, zfs_dataset)

This function setup the step structure with the basic information needed o create a step.

Inputs:
message: Message to print out at start of the step. i.e. "Downloading IPS packages"
name: User friendly name for the step
zfs_dataset: Name of the zfs dataset used when the snapshot is taken

Return code:
None

8.2.3 DC_delete_checkpoint(cp, step)

Delete a designated checkpoint indicated by the step object. It will destroy the corresponding zfs snapshots and delete the .step file. At the beginning of a full build, all checkpoints should be deleted. If you start a build with the -r or -R flags, checkpoints from that stop onwards should be deleted. This prevents files from the previous build from interfering with the current build.

Inputs:
cp: the object containing all checkpoint related information
step: checkpoint number to be deleted

Return code:
None

8.2.4 DC_execute_checkpoint(cp, var)

This function will check to see if the current step should be executed. If the step to resume at is less than the current step, a checkpoint will be created with the designated name. This name must be the same as is specified in the corresponding call to step_setup() that occurs in DC_checkpoint_setup(). If the current step is the step to pause at, a snapshot (s) is taken and execution is stopped. If the current step is the step to resume at, a zfs rollback of the datasets is done. If the current step is later than the resume step (or no resume step is specified) , a zfs snapshot of the specified datasets will be taken. The name of the snapshot is <dataset_name>@.step_<name>.
Calls to DC_execute_checkpoint should be put wherever a checkpoint is to be taken during the build process.

Input:
cp: the object containing all checkpoint related information
var: A value that can either be an int or a string. If it is an int, it is the number coorsponding to a checkpoint. If it is a string, it is the checkpoint name. This value is used to determine which step will be executed.

Return Values:
0: executes the step without any error
1: don't execute the step
-1: stop execution on return

8.2.5 DC_verify_resume_step(cp, num)

Verify that the step specified to resume at is valid. A valid step is less than or equal to the step at which the previous build exited. This will be determined by checking for the existence of the

.step<name> file that corresponds to the number of the checkpoint. If the user specifies an invalid step to resume at, an error message will be printed and execution will stop.

Input:
cp: the object containing all checkpoint related information
num: step number the user specified to resume at

Return Values:
0: success
-1: error

8.2.6 DC_remove_state_file(step)
Removes the file .step<name> specified in the step object

Input:
Step object indicating the step to remove.

Return Values:
None

8.2.7 DC_verify_resume_state(cp)
Verify that the state of the resumed run is consistent with the previous run. If the states disagree, a USEFUL warning message will be printed but execution will continue.
To verify the state, the saved .step<name> manifest will be compared to the manifest specified on the command line. This will be done for the step resumed at and for every step that comes before it.

Input:
cp: the object containing all checkpoint related information

Return Value:
None

8.3 Data structures

8.3.1 class step:

```
step_num = 0
_step_num
_step_name
_message
_zfs_snapshot
_state_file

def _get_step_num():
def _get_step_name():
def _get_step_message():
```

```
def _get_state_file():
def _get_zfs_snapshot():
def __init__():
```

8.3.2 class checkpoints:

```
step_list = []
_resume_step
_pause_step
_current_step
_manifest_file
_checkpoint_avail
_zfs_found
_pkg_image_mntpt
_pkg_image_dataset

def incr_current_step():
def get_current_step():
def get_resume_step():
def set_resume_step(num):
def get_pause_step():
def set_pause_step(num):
def get_manifest():
def  set_manifest(manifest):
def set_checkpointing_avail(flag):
def get_checkpointing_avail(flag):
def set_zfs_found(flag):
def get_zfs_found():
def set_pkg_image_mntpt(mntpt):
def get_pkg_image_mntpt():
def set_pkg_image_dataset(dataset):
def get_pkg_image_dataset():
def create_checkpoint(name):
def step_setup(message, dataset_list, name):
def __init__ ():
```

9 Manifest

The manifest is a template or blueprint which defines how to create a distribution, and thus defines the distribution itself. Initially, users will edit this file by hand. Eventually, tools will be provided to generate and/or edit the file.

9.1 Format

9.1.1 Style

The manifest will be an XML document. There are several benefits. The nested name-value pair

format of XML helps lay out details well and associate them in an organized way. Name-value pairs help to foolproof the data. Properties can be added in any order because the name is listed along with the value. Properties can be omitted (and either caught as omitted and trigger an error, or can be given default values). Name-value pairs aid extensibility by being explicit. Syntax can be checked. XML may be more wordy, but it's wordiness helps prevent mistakes and gives many other benefits.
Why not a flat file? XML differs from flat files by explicitly nesting properties inside of the items they affect. Flat files would be simpler but more prone to making mistakes; while lack of nesting would make flat files simpler, it would require a formatting method which would be more prone to mistakes. For example, suppose that instead of having to spell out a nesting, one could just tell the nesting level by indentation: the more indentation the greater the nesting level. It would be an error if one forgets to indent or indents too much, which would be easier to do than to mis-enter a property with XML.
Another alternative is Enhanced SMF Profiles. These are not selected because SMF has not been created to serve our proposed purpose. SMF is intended for starting and maintaining a list of running services, not for setting up a distribution. SMF could be used, but would be overly complicated. Different configuration details would be handled by different services, each in different profiles, and SMF would kick off these services. This would only need to be done once. Since we would need to set up the scripts for the services anyhow, why not just call them directly without the need for SMF? SMF has been placed on the table because it supports layered profiles, but an XML parser can be modified to replace a value of a previously found name (of a name-value pair) if a new value is found.
The verbosity which brings about XML's biggest advantages also present its main disadvantage: that XML is hard for humans to read. A second easier “language” was considered with keeping XML in the background, but having two languages would be confusing. The complexity of XML is mitigated somewhat by delivering the proper tools to build and read XML files. The benefits outweigh the disadvantages.

9.1.2 Design Considerations

While an element starts off as having a specific type or purpose(eg: stderrlog_name), the grammar used to parse and interpret that element ultimately sees it as a generic type (like “string”). Context, or what an element represents, gets lost as its type gets more generically interpreted. This impedes verification. Some solutions to this problem are:

- Add context by adding context-defining elements to the element chain. For example, adding a “filename” element along the element chain just ahead of stderrlog_name can tell the user of stderrlog_name that it is a filename.
- Add context by adding a “type” attribute to the stderrlog_name element, in effect marking that element to be interpreted as a filename.
- Using a second XML file, map elements that exist at certain positions in the data tree as certain types. This has the following advantages over the previous two solutions:
  - The second XML file does not need to be known by the user. This removes complication from the user's point of view.
  - The XML file which the user uses does not need to contain the extra cruft which gives the added context.
  - This approach is more flexible, as the second XML file can define how to process the marked elements, rather than just mark them.

9.2 Content

The content of the manifest is either used by the Distribution constructor to set up it's work environment or control it's program flow. The content is also used to define the content and how an image is constructed.
The manifest is designed in such a way that other content needed for specific finalizer scripts can be added easily.

9.2.1 General

These parameters deal with how the distribution constructor works.

- **Distribution Name**: String.
  - Name of the distribution.
  - Default is YYYYMMDD-HHMI

- **Stop-On-Error Flag: B**oolean.
  - True means "Stop when an error is encountered." Defaults to True
  - When clear, DC forges ahead even when errors occur from the DC application and the finalizer scripts.

- **Checkpoint-enable flag: B**oolean.
  - True means "Enable stopping at checkpoints." Checkpointing is not performed if set to False.
  - Default is True if the build area is a ZFS dataset
  - This value has no effect if the build area is not a ZFS dataset. Checkpointing will be off.

- **Which checkpoint to start from**: Unsigned integer.
  - If left blank, start at the beginning. If no such checkpoint, log an error.

- **Build Area**: String.
  - Specifies the work area used for the image construction.
  - Can be a path or a zfs dataset name
  - Defaults to rpool/dc-bld
  - See section 7.2 for more details on the structure of the build area

9.2.2 Image Parameters

The following parameters are commonly used for most bootable image types, such as the Live CD, Text Installer Image, and the Automated Installer Image.

- **Preferred Package Repository Authority**: String.
  - Preferred pkg(5) repository to install packages from.
  - Mirrors to the preferred authority may be specified.
  - Format of the strings representing the authority and it's mirrors must confirm to the format specified by pkg(5)
  - Cannot be left blank.
  - Default URL is: http://pkg.opensolaris.org/release

    - Default authority name: opensolaris.org

- **Additional non-preferred Package Repository Authority**: String.
    - Additional non-preferred authorities to pull packages from. Multiple additional non-preferred authorities maybe specified.
    - Mirror(s) can be specified for each additional non-preferred authority
    - Format of the strings representing the authority and it's mirrors must confirm to the format specified by pkg(5)
    - This is optional.
- **Post install preferred authority:**
    - Preferred authority and it's mirrors for the system to use after the install has completed
    - This is optional. If not specified, the repository used during image construction is used post-install

- **Post Install non-preferred package repository authority:**
    - Additional authority and it's mirrors for the system to use after the install has completed.
    - More than one post install non-preferred package repository authority may be specified.
    - This is optional.

- **Package List**: Package FMRI
    - Specifies the list of packages needed to build the image.
    - Each entry on the list should be 1 FMRI
    - The FMRI specified must comply with the requirements from pkg(5)
    - Can not be left blank

- **Post Install Remove Packages**: Package FMRI
    - Specifies the list of packages to remove after the specified package list is installed.
    - Each entry on the list should be 1 FMRI
    - The FMRI specified must comply with the requirements from pkg(5)

- **Generate IPS Search Index**: Boolean
    - Indicate whether the IPS index should be generated for pkg install and pkg uninstall.
    - IPS Search Indexes are extremely large files. They takes up a lot of space in an image.
    - Default is False, which is to not generate the IPS search index.

- **Hostname**: string
    - Hostname to use in the output image.
    - Defaults to “opensolaris” if not specified.

- **Grub Menu Entries**
    - Entries that make up menu.lst
    - By default, the first line of /etc/release is used as the title for grub menu entries. If a special title is needed, users can specify the title in the <title> tag.

- **Bootroot contents list**:
    - List of files and directories to be in the bootroot. The bootroot contains the minimal list of contents in order to be able to boot and setup a running system.

- All specified files and directories are taken from the pkg-image area. They must exist in the pkg-image area.
    - Users who wish to include everything in a directory except a couple files can specify the directory they want to include, then, specify to exclude a selected set of files.
    - On certain platforms, such as SPARC, per-file compression is done. This can be disabled on individual files which are read in as part of a directory of files, by base_including that file and specifying fiocompress=”false” for it.

- **Bootroot compression**:
    - Indicate the compression type. Acceptable values are gzip or none.
    - For gzip compression, a numeric level between 0 and 9 can also be specified.
    - Gzip is the default compression type used.
    - No compression will take place if type is none or level is zero

- **Finalizer script list:**
    - List of scripts to be called for performing the actual construction of the image.
    - Each list element will contain strings for module name and arglist.
    - See section 10 for further details on Finalizer scripts.

10 Finalizer Scripts

Finalizer scripts are the set of user provided programs the Distribution Constructor execute to create the image. The use of these scripts by DC allows DC the flexibility to create any image. The user specifies the full path to the program to be executed, the order in which they should be executed, as well as arguments to be passed into the scripts in the manifest.

Finalizer Scripts can be any elf binaries, shell scripts, or Python programs. Supporting data used by the scripts can be passed into the scripts as arguments. Supporting data can also be specified in the manifest. For example, the script to populate the image package area with the list of specified IPS packages reads the package list from the manifest.

Finalizer scripts are separate processes started by the DC application. As a result, they do not have access to the memory space of the DC application, which is where the values from the manifest are loaded. In order to allow the DC application and the finalizer scripts to access the information from the manifest, DC starts a server process which makes all the values in the manifest available. Detailed description of the server process is discussed in section 12.5.1. All finalizer scripts access the manifest values via this server. A convenient utility is provided so scripts can easily get the manifest values. The utility is called ManifestRead, and it is discussed in detail in section 12.5.4

By default, the DC passes 5 arguments to all programs/scripts it executes. These argument contains information that's needed by all scripts. Any user specified argument to finalizer scripts will start as argument 6. The 5 arguments passed to the scripts are:

1. **server socket file name**: server socket used for accessing manifest data
2. **package image area path**: path to the package image area
3. **tmp dir**: path to the /tmp directory for temp files.
4. **Bootroot build area**: this is where the bootroot gets put together.

5. **Media area**: this is where the output images will go

Some of the steps for creating images are common for most image types. For example, installing a specified set of packages from one or more IPS repositories, or creating and configuring the root archive with a set of files. The DC project will provide a set of scripts for these common operations, which users can use directly for building images. The list finalizer scripts provided by the DC is discussed in section 15.

11 Execution Engine

The execution engine coordinates calling user specified finalizer scripts to create the resulting image. This is initialize by the DC application after the manifest is parsed. All finalizer scripts, and their arguments will be registered into the execution engine, and the execution engine will execute the scripts in the order that they are registered.

11.1 Programming Interfaces

11.1.1 __init__(first_args_list=None)

The Python class is DCFinalizer. It is possible to instantiate multiple instances if desired, to provide for general case use. The Distribution Constructor, however, uses only one instance. The argument accepted here is a list of arguments which will be always be passed to all scripts called by the finalizer, unless it is None or not specified. Arguments registered with register() will be passed as the higher-numbered arguments. This argument is intended to accept any arguments most scripts will need, including a socket name to connect to get XML manifest data, a pointer to the pkg_image area, and so on.

11.1.2 register(module, arglist=())

Register a module or python method call with the finalizer. A module can be a script, a binary or a method out of a python module. The module type is determined automatically, as if it is run in a shell directly, so it is important for scripts that their first line reflect the shell in which they is to be run.
Modules are queued for execution in the order they are registered.

Args:

- ***module*** (string) is the path of the script, program or python module. This must be provided.
  Note: to call a single method in a python module (instead of invoking the module itself), specify *module:method*
- ***arglist*** is the list of arguments (strings or numbers) to pass to *module*. It may be an empty list or not specified if there no arguments. Note that if an argument was passed to the finalizer's constructor, that argument will always be passed as the first argument to the script and all arguments provided here will be shifted over by one.

Returns:

1 (general error): A Python module, shell script, shell interpreter or binary are not runnable.
Returns 0 (success) otherwise.

11.1.3 change_exec_params(output=None, error=None, stop_on_err=None, logger_name=None)

Change the logfiles and error handling environment in which modules are executed. Calls to ***change_exec_params()*** may be interspersed between calls to ***register()***. A call to ***change_exec_params()*** made before a call to register a particular module will queue an environment change which will take effect just before that module is later executed.
Args:

- ***output*** (string) specifies where stdout is to go. This can be a filename, an AF_UNIX socket pathname, ***stdout*** to go to the console, or ***None*** meaning no change. No advance checking is done that the file can be created or written. This argument is not used by DC. It's provided for other application.
- ***error*** (string) specifies where stderr is to go. This can be a filename, an AF_UNIX socket pathname, ***stderr*** to go to the console, or ***None*** meaning no change. No advance checking is done that the file can be created or written. This argument is not used by DC. It's provided for other application.
- ***stop_on_err*** (boolean) specifies whether or not the finalizer continues on to the next module, after a module it called has encountered an error. An error is defined as a return status of nonzero and/or a raised exception.
- ***logger_name***(string): Name of the logger to use. Passing None means we don't want to logging of the command's stdout/stderr. If a logger name is specified, it is an error to also specify file names of output or error.

Returns:

1 (general error): One or more arguments are invalid.
0 (success) otherwise.

11.1.4 execute()

Execute the parameter change requests and module execution requests in the order they were registered or enqueued. If ***stop_on_err*** is set (see ***change_exec_params()***) when an executing module returns an error or an exception is raised, quit and return the error status. If ***stop_on_err*** is not set, continue through the list of registered modules. If multiple errors occur, return the value of the first error encountered. If multiple exceptions are encountered, save the first one; it may be retrieved by calling ***get_exception()***.

Returns an ***errno*** value returned by an unsuccessfully run program, script or python method.
Returns ***-signum*** if a signal was caught by a program, script or python method.
Returns 1 if an exception was raised in a called python module or method or when trying to invoke an environment to run a program or shell script. Use the return value from ***get_exception()*** to tell whether an exception was raised, and thus to distinguish whether the return value is due to an exception or is an errno value of one (EPERM).
Returns 0 if all registered modules returned successful status.

11.1.5 get_exception()

Get the handle to the first exception object encountered during an ***execute()*** run.

Returns the handle to the first encountered exception.
Returns ***None*** if no exceptions were encountered or ***execute()*** has not been called.

12 Manifest Parsing Module

12.1 Parser Type

There are two classes of XML parsers for Python: SAX and DOM. SAX, or "Simple API for XML", is a simple parser which hands off the bits and pieces of an XML document as it reads them, to user-written callbacks which can process the data as they see fit. It is up to these callbacks to build any internal data structures to store them. DOM, or "Document Object Model", builds an internal tree containing the data and provides functions to extract the data.

SAX is very flexible in that it leaves to the application to decide which XML information gets used. The application can optimize memory usage and performance based on how it stores the data. It is more complicated to use, however, for precisely the same reasons: the application has to do more to store the data.

The flip side of this is DOM. The main complaint of DOM is that it can use lots of memory for large XML files, because it creates its own data tree, and stores all of the data it reads.

Of SAX and DOM, DOM better suites our purpose. DC's manifest is bounded and fairly small, so memory won't be an issue. DOM also provides routines to manipulate the data tree, allowing for layering. Several manifests can be read in, and their data updated/appended in a main data tree. Finally, DOM allows for writing out a tree into a new XML file. This allows for a tree to be customized, and a new XML file containing all customizations to be written and saved.

12.2 Schema Type

A RelaxNG schema will be used for validating XML data. It was evaluated along with DTDs and W3C schemas and was chosen for the following reasons:

- It is delivered with Solaris.
- Validation works. (W3C schema validation is incomplete as of this writing.)
- It does better type and pattern checking than DTDs.
- They are easier to read than DTDs (as they use a nested XML-like syntax).

12.3 DOM Tree Access Abstraction Layer

Python provides the facility for setting up the DOM tree, but tree access is primitive in that a node with a particular “path” from the root cannot be easily searched for. This project proposes a new module which fills this need. Desired nodes are specified via a “nodepath” from the root, or from some other part of the tree. It can search, add a node in a particular place, and replace the value of a particular node. If a search from the tree root would give multiple nodes, one can “zoom in” at a more specific location of the tree to get a specific node. Alternatively, the search may be narrowed by specifying values of nodes along the nodepath if they are known. The class which abstracts the tree and provides this functionality is called TreeAcc. Along with TreeAcc is a class of object called TreeAccNode

which abstracts a tree node, making the information easier to get. Together, these classes are used for setting up the data (including setting it up in phases, or layering), validating it and retrieving it.

12.3.1 TreeAcc Class

The TreeAcc class allows node search, node add and node value-replacement functionality. It also has a method to save the tree to an XML file.
Most methods take a nodepath. This argument represents a slash-delimited string, much like a Unix pathname, which details the elements in the tree that must be traversed to get to the desired node(s). The name of the root node is not included. ".." may be used to go up the tree if needed; this is convenient if starting in the middle of the tree.
The node names which make up the nodepath may include attributes. (Attributes may only be the last branch in a path.) Paths to attributes are specified the same way as paths to elements.
Nodepaths may optionally include values or attributes of elements traversed along the way, in order to narrow the search. Specify a value along the way with the following syntax:

.../element_name=value/...

Searches may be narrowed by matching elements with children elements (or grandchild attributes) of specified values, even if those specified values are not along the direct path to the values desired, as per the following examples:
Narrow a search to include children of elements containing attributes of a certain value:

.../element_name[attribute_name=value]/...

For example, if the password for all users are found by specifying:

"live_img_params/user/password:

then the following will find the password for the user with a username attribute of "joeblow":

"live_img_params/user[username=joeblow]/password"

More sophisticated searches are possible:

– In the above example, "username" is the attribute immediately below "user", but it could be a nodepath extending from "user" consisting of multiple elements. If the above example is represented by "a/b[c=cval]/d", "a/b[c/e/f=fval]/d" would also be a valid nodepath.
– Multiple limiting values can be specified for a single element. Use a colon to separate them. So, for example, "a/b[c/e/f=fval:g/h/i=ival]/d is a valid nodepath.

Note: the method definitions below are slightly paraphrased for clarity. For example, they don't show "self" arguments required of python instance methods.

12.3.1.1 __init__(xml_file)

The constructor takes the name of an XML file as an argument. It then instantiates the DOM tree and its TreeAcc representation.

12.3.1.2 find_node(path, starting_ta_node)

Search for node(s) which match the path. If *starting_ta_node* is specified, start the search from that node; otherwise start from the tree root. For example, given the following snippet of XML code:

```
<distribution>
	<live_img_params>
		<user username=jack>
			<homedir=/export/home/jack>
```

<user username=jill>
<homedir=/export/home/jill>

The following would return a list of the two user nodes:
nodes = find_node("live_img_params/user")
The following would return a list of the two username (attribute) nodes:
nodes = find_node("live_img_params/user/username")

If no nodes are found, find_node returns an empty list.

12.3.1.3 replace_value(path, new_value, starting_ta_node)

Search for a node which matches the path. If *starting_ta_node* is specified, start the search from that node; otherwise start from the tree root. Err out if more than one node matches or if no match is found.

12.3.1.4 add_node(path, value, type, starting_ta_node, is_unique)

Add a node with path *path* of type *type* (either element or attribute) relative to *starting_ta_node*. Path is relative to the root if *starting_ta_node* is not specified.
If *is_unique* is true (default), and there is already a node which matches *path* then err out. *is_unique* must be specified when adding attributes.
If there is more than one matching parent node (or node with the same path less the final branch), err out as where to put the new element is non-deterministic. If *is_unique* is false, add the element even if one or more elements which match already exist (again, assuming the location is deterministic).
Value may be *None* for elements. Err out if *None* is specified for an attribute.
This method returns a TreeAccNode that corresponds to the added node. This node is useful for adding attributes to a just-added element. For example, suppose *add_node()* has just added a new user node (call it *new_node*) alongside other existing users. Specify it as the starting node when calling *add_node()* to add a username attribute to it, to satisfy that the new attribute location is deterministic.
For example:
new_node = tree.add_node("live_img_params/user", None, tree.treeroot_ta_node, False)
tree.add_node("username", "jack", new_node)

12.3.1.5 save_tree(out_file)

Save the DOM tree to out_file. Err out if the output file cannot be opened or written.

12.3.1.6 get_tree_walker()

Return an obtuse TreeWalker object. This object is passed repeatedly to *walk_tree()* to return all elements and attributes of the tree.

12.3.1.7 walk_tree(walker)

Return a list of TreeAccNodes which corresponds to a single element and its attributes. Takes a TreeWalker object as an argument. Modifies the TreeWalker object state so that when the same TreeWalker is passed again in a subsequent call, a new list corresponding to another element and its attributes is returned. Returns None when the tree walk is complete.

12.3.2 TreeAccNode Class

Objects of this class represent nodes in the DOM tree. It contains only data and accessor methods.

Classbound constants it defines:

- TreeAccNode.ATTRIBUTE: attribute node type
- TreeAccNode.ELEMENT: element node type

Instance data it defines:

- name: Used as part of a path, this is used as the DOM element or attribute name.
- type: Must be one of TreeAccNode.ATTRIBUTE or TreeAccNode.ELEMENT. These types correspond directly with their DOM tree counterparts.
- value: String value represented by this node. May be set to *None* for elements if there is no value.
- attr_dict: Dictionary of attributes if this node is an element. Not used (and set to None) for attribute nodes.
- element_node: Corresponding DOM tree element node represented. For objects representing elements, this corresponds directly to the DOM tree element node represented. For objects representing attributes, this corresponds to the parent (element) node to which the represented attribute is attached.

12.3.2.1 __init_(name, type, value, attr_dict, element_node)

Constructor. Intended for use by the TreeAcc class; not normally called by end users.

12.3.2.2 get_name()

Returns the name string of this node.

12.3.2.3 get_path()

Returns the string path of this node (back to the tree root).

12.3.2.4 get_value()

Returns the value string of this node.

12.3.2.5 get_attr_dict()

Returns the attribute dictionary of this node. Returns None if this node represents an attribute.

12.3.2.6 get_element_node()

Returns a reference to the corresponding DOM tree node if this node represents an element, or returns a reference to the parent DOM tree node if this node represents an attribute.

12.3.2.7 is_leaf()

Returns True or False that this node has no children. For element nodes, returns True if that element has no child element nodes. Always returns true for attribute nodes.

12.3.2.8 is_attr()

Returns True or False that this node represents an attribute.

12.3.2.9 is_element()

Returns True or False that this node represents an element.

12.4 Manifest Pre-Processing

An XML schema can go only so far to validate an XML document. A schema does no setting of defaults for elements, and a RelaxNG schema does no setting of any defaults. The Distribution Constructor does pre-processing of the manifest before using it to insure a complete specification and a robust distribution. The Distribution Constructor supports semantic content validation and setting of defaults.
For the purposes of this document, unless otherwise stated, *nodes* refer to TreeAccNodes, not DOM nodes. This means there are two kinds of nodes: element and attribute nodes; and both kinds of nodes have self-contained values.

12.4.1 Defaults/Validation Manifest

The preprocessing itself uses an XML specification to define its actions. This makes it flexible and versatile. This specification will be called the defaults/validation XML manifest, or "defval manifest." It maps a value or a method (to use for validation or calculation of a default) to a given node or set of nodes in the DC specification manifest (called the DC manifest hereafter).

Preprocessing can call external methods to calculate a default for a node or to validate a node. The first section of the defval manifest declares these methods, which are called helpers, to the rest of the defval manifest. Each helper specification has attributes for method name and the module that method is delivered in. Each helper also has a nickname or refname by which other parts of the defval manifest easily refer to that helper. The defval manifest distinguishes between default-setter helpers and validator helpers.

12.4.1.1Helper Methods

Helper methods take one argument which is a TreeAccNode, so they can get at the data they need. The argument to default setter methods is the parent of the node in which to set the default. The argument to validation methods is the node being validated. Often the DC manifest tree needs to be parsed for some value needed in the calculations; the tree can be retrieved from the node argument.
Modules with helper methods must be accessible and importable by python, and the methods therein must be defined in a class with the same name as the module.

12.4.1.2Defval Manifest vs DC Manifest

The defval manifest must be kept in sync with the DC manifest schema. If the DC manifest schema should change, new nodes could be introduced which might need validation or defaults and the defval manifest needs to accommodate the new requirements. Likewise, validation could break or default requirements could change if elements or attributes are changed in the DC manifest schema.
Note that the defval manifest, defval schema and DC manifest schema must be kept in sync with each other. They are delivered as part of DC and are not to be changed by end users. End users will only be expected to change the DC manifest in order to change the specification of their distribution. The data will be validated and defaults processed under the same DC manifest constraints as defined by the DC manifest schema.

12.4.2 Pre-Processing Sequence

Here is the sequence of actions for pre-processing:

- Validate the defval manifest. Validate it using xmllint against its RelaxNG schema.
- Create the (main) DC manifest (in-memory DOM) tree by reading in the DC manifest.
- Layer data from any secondary DC manifests in the in-memory tree.
- Perform defaults processing to fill in any missing but required items.
- Perform validation processing.
- Write out the in-memory tree to a file. This file can be saved and used as input in subsequent iterations or uses of the Distribution Constructor, can be used to debug issues with the manifest, and will be used in final validation.
- Validate the just-written manifest against the DC manifest RelaxNG schema using xmllint. If successful, the still-in-memory data is valid and ready to be used by other parts of the DC to build the distribution.

12.4.3 Defaults Processing

Specification in the defval manifest of which defaults to check in the DC manifest are given by *default* elements. Such elements contain the pieces specified below.

Defaults processing throws a ManifestProcError exception if an error occurs anytime during processing, after all defaults processing is complete.

12.4.3.1Value

The element's value is either a value to use as a default, or the refname of a method by which the default value is calculated, depending on the *from* attribute.

12.4.3.2from attribute

The *from* attribute can be set either to *helper* or *value*. It defines what the element's value contains.

12.4.3.3path attribute

The *path* attribute defines the node(s) for which to set the defaults. Recall that nodes are specified by a path and an optional starting node, and that a single path and starting node can match several nodes.

12.4.3.4type attribute

The *type* attribute refers to the target node's type and must be set to either *attribute* or *element*. Processing of elements and attributes is different, and the node type is needed when missing nodes are created as part of defaults processing.

12.4.3.5missing_parent attribute

Nodes which are missing are created if their parent node exists. An optional *missing_parent* attribute specifies what to do if the parent node is also missing. *Missing_parent* can be set to one of the following:

- *create* – Create any nodes between the tree root and the node to contain the default. Use this in cases where a node and its parents must exist and the system can calculate a default value for the node. No defaults will be set for any parent nodes.
- *skip* – Do nothing if the parent node is missing. Use this in cases where a default value must be set if the immediate parent node exists, but nothing is needed if the parent does not exist. An example of this would be to set a default home directory for each user that exists. This might be modelled in in the DC manifest with a path like *.../user/homedir*. If there are no nodes which match *.../user* (i.e. no users exist) then there is no need to create *homedir* nodes.
- *error* – Fail if the parent node is missing. Use this when both the parent and target nodes must

exist and contain a default. This is the default setting for when the *missing_parent* attribute is not specified.

12.4.3.6 skip_if_no_exist attribute

There may be large chunks of the tree which are inappropriate for setting defaults in certain circumstances. For example, suppose the tree had an large optional section. It would be inappropriate to create a chain of nodes there if that section has been omitted, even when *missing_parent="create"* is specified for a node in that section. An optional attribute *skip_if_no_exist* specifies to skip processing or setting a default when a given starting portion of a path (e.g. to the root of the tree's optional section) is matched. For example, *skip_if_no_exist="optional_section"* completely bypasses checking and setting of a default if the *path* begins with "optional_section".

12.4.4 Validation Processing

Validate elements come in three varieties: *path*, *group* and *exclude*. Validation continues through single failures, and throws a ManifestProcError exception after all validation is complete, if any failure is found.

12.4.4.1 Validate Path Processing

A v*alidate path* element is similar in structure to a *default* element. It has a *path* attribute to specify the node(s) to validate. Its value is a (commas-optional) list of validator helper method refnames specifying the methods used to validate the node(s).
What to do with missing nodes is specified by the optional *missing* attribute, which can be set as follows:

- *ok* – Do nothing for a missing target node and do not fail validation because of it. Use this in the case of an optional node.
- *ok_if_no_parent* – If a target node is missing and its parent is also missing, do nothing and do not fail validation because of it. Fail validation though, if a target node is missing and its parent is present.
- error – Unconditionally fail validation if a target node is missing.

12.4.4.2 Validate Group Processing

A *validate group* element consolidates nodes of several paths to be validated by a single method. *Validate group* elements have a *group* attribute which is set to the refname of the validation method. The element contains for its values a (commas-optional) list of pathnames to validate. Missing nodes do not fail validation.

12.4.4.3 Validate Exclude Processing

A *validate exclude* element runs a validation method on all nodes of the tree unless excluded. It has an *exclude* attribute which is set to the refname of the validation method. It contains for its values a (commas-optional) list of pathnames to exclude from validation. Missing nodes do not fail validation.

12.5 Manifest Server and Reader Interfaces

A set of higher-level interfaces are provided for reading and setting up of manifest data. Data is made available to both the project's main (the DC proper, in the case of DC) and finalizer scripts using string-based retrieval interfaces. Interfaces are implemented by the manifest server and manifest reader

objects ManifestServ and ManifestRead, and by commandline interfaces of the same names which instantiate and use these objects.

Finalizer scripts read data through manifest reader objects and commandline interfaces. Shell scripts and programs called from the finalizer run the ManifestRead commandline interface which instantiates a ManifestRead object to fetch the data. Python scripts and methods called from the finalizer instantiate a ManifestRead object directly.

The ManifestServ module is called by the DC proper (or project main) to perform initialization (validation, setting of defaults, creation of the data tree), to start a server that provides data to instances of manifest readers, and to read data for itself.
Each of these will now be described in more detail.

12.5.1 ManifestServ class

ManifestServ objects provide the project's main with easy-to-use interfaces for initializing and retrieving data. Initialization occurs when these objects are instantiated, and provides a number of options. Methods for reading the data and for starting the server so that ManifestRead objects can access the data are also provided.

12.5.1.1 __init__(manifest, valfile_base=None, out_manifest=None, verbose=False, keep_temp_files=False)

This validates and initializes all XML data for retrieval from an in-memory tree. Validation and initialization consists of the following steps:

- Initialize and validate defaults/content-validation tree.
- Initialize the project data (manifest) tree.
- Add defaults to the manifest tree.
- Validate semantics/content of manifest tree.
- Validate the manifest tree against the manifest schema.
- Optionally save a nicely-formatted XML file containing all adjustments. This is the output_manifest. Name is of the form: "/tmp/<manifest_basename>_temp_<pid>

This method requires that the manifest, manifest schema and defval manifest files be in place with specific names as described in the argument descriptions below.
Args:
●***manifest*** (string) represents the full pathname of the manifest file. (The *.xml* suffix is optional in the argument, but is required of the manifest file itself.) The manifest file contains the base XML data, which is validated, appended with defaults, and then made available for query.
If not overridden by the *valfile_base* arg, the base of *manifest* arg represents the basename of the schema and defaults/content-validation manifest files needed during initialization. Their names, relative to the *manifest* argument, are as follows.
Given a *manifest* argument of */filepath/manifestname.xml*:

- schema will be named */filepath/manifestname**.rng***
- defval manifest will be named */filepath/manifestname**.defval.xml***

  The base names of the schema and defval manifest will be overridden to be *valfile_base* if *valfile_base* is provided.

●***valfile_base*** (string) if provided, overrides the base name of the schema and defval manifest files.
This allows the schema and defval manifest to be provided, for example, as part of a project and placed

in one area, while the manifest can be provided by the user and placed somewhere else. This functionality is used by DC but is optional as some projects may not want to use it.
●***out_manifest*** (string) specifies the name of a nicely-formatted output manifest file containing all of the data changes (e.g. defaults added) made during initialization. This file can be useful for debugging, and also for saving for future use as an input manifest.
●***verbose*** (boolean) enables on-screen display of defaults, content validation and schema validation. Defaults to False.
●***keep_temp_files*** (boolean) leaves temporary files around for inspection after termination. This may be useful for debugging. Temporary files will be in the */tmp* directory and will be keyed off the *manifest* arg.

Returns:

A ManifestServ object whereby fully initialized and validated XML data can be retrieved.

Note that exceptions can be raised at any stage of initialization. See the appropriate sections in this document for more information on specific exceptions for specific stages.

12.5.1.2get_values(request, is_key=False)

This retrieves data from the manifest. Note that more than one datum may be returned per request, if a request matches multiple nodes.
A convenient shortcut for keys is supported. Keys have a special place in the manifest and *get_values()* knows where to find them without requiring their full nodepath to be specified. This makes key lookup easy to use.
Keys are general and so are kept in their own separate manifest section. Manifests for every project can place their keys in the same section. Keys are looked-for at the level immediately below the root level in the tree, as illustrated:

```
<distribution name=...>        <!-- tree root for DC -->
  ...
  <key_value_pairs>
    <pair key="key1" value="value1">
    <pair key="key2" value="value2">
    <pair key="key3" value="value3">
  </key_value_pairs>
</distribution>
```

Args:
●***request*** (string) specifies where to look for data to retrieve; it is a key when *is_key* is *True;* otherwise it is a nodepath in the XML tree. Note: what is in the node is what is returned. If the node holds a string with a list of items, the string is returned as stored.
●***is_key*** (boolean) specifies whether *request* represents a key or is a nodepath. When *True*, then *request* is a key.

Returns:

A list of data from nodes matching the request, one datum per matching node. Returns an empty list if no matches are found.

12.5.1.3 start_socket_server(debug=False)

Start the socket server so that ManifestRead objects can connect to it and fetch data. This mechanism is needed because scripts invoked by the finalizer may want to get at data which has been set up by the project, but finalizer scripts are not part of the same namespace or process space as the project's main. The server creates an AF_UNIX socket usable by other processes on the same system. The name of this socket will be of the form */tmp/ManifestServ.<pid>* where *<pid>* is a process ID. The project must call *stop_socket_server()* to cleanup the socket.
This method does not return anything, but raises socket exceptions if it encounters problems.

Args:

- ***debug*** (boolean) enables debugging messages (regarding protocol, etc) during socket utilization.
-

12.5.1.4 stop_socket_server()

Stop the socket server and cleanup the socket.
This method does not return anything, nor does it raise any exceptions, even when the server isn't running.

12.5.1.5 get_sockname()

Get the name of the socket. The retrieved socket name may then be passed to finalizer scripts or other programs running outside of the project's namespace and process space which need it to get access to the XML data via ManifestRead objects or commandline interfaces.

Returns: The name of the socket.

12.5.2 ManifestRead class

Objects of this class are used by processes running outside the namespace and process space of the main project, to retrieve data from the manifest set up by the project's main. It has the following methods:

12.5.2.1 __init__(sock_name)

Instantiate a ManifestRead object which will communicate through the socket named *sock_name* to retrieve data from a ManifestServ object.
Args:

- ***sock_name*** (string) is the name of the socket to use to connect with a ManifestServ object.

Returns:

an initialized ManifestRead object.

This method can raise socket exceptions when attempting to create or connect a listener socket.

12.5.2.2 get_values(request, is_key=False)

This retrieves data from the manifest. The method returns one datum per matching node. Note that more than one datum may be returned per request, if a request matches multiple nodes.
Although this method is for remote access, the syntax and semantics (except exceptions) of this method are the same as for the ManifestServ method of the same name. Please see the ManifestServ version for a description of keys.
Args:
request (string) specifies where to look for data to retrieve; it is a key when *is_key* is *True;* otherwise it

is a nodepath in the XML tree. Note: what is in the node is what is returned. If the node holds a string with a list of items, the string is returned as stored.
●***is_key*** (boolean) specifies whether *request* represents a key or is a nodepath. When *True*, then *request* is a key.
Returns:
A list of data from nodes matching the request. Returns an empty list if no matches are found.
This method can raise exceptions due to socket errors.

12.5.2.3print_values(request_list, are_keys=False, force_req_print=False)

Retrieve data from the manifest and print it to stdout. Multiple requests may be processed with one call. If *are_keys* is *True* then all requests are keys, else they are all nodepaths.
Args:
●***request_list*** (list of strings) is the list of requests to process. Requests are keys when *is_key* is *True;* otherwise they are nodepaths in the XML tree. If the node holds a string with a list of items, each item in the list is printed on its own line.
●***are_keys*** (boolean) specifies whether *requests* are keys or are nodepaths. When *True*, then *requests* are keys.
●***force_req_print*** (boolean) specifies whether the request prints alongside each result, if there is only one request. When multiple requests are made with one call, the request always prints alongside each result.
Returns:
Nothing. Output is printed to the screen.
This method can raise exceptions due to socket errors.

12.5.2.4set_debug(on_off)

Turn on or off debug messages regarding protocol.
Args:
●***on_off*** (boolean) Enables messages when *True*, disables when *False*
Returns nothing and raises no exceptions.

12.5.3 ManifestServ commandline interface

From a ManifestServ object's point of view, this commandline interface performs the function of a project's main program. Useful for testing, this program instantiates a ManifestServ object, can start up the socket server, and can get data locally.
The program enters a loop to accept requests for data, after it is done initializing. The loop operates in two modes. The loop accepts nodepaths by default, but can be switched to accept keys by entering "+key". When in key mode, the loop can revert back to accepting nodepaths by entering "-key".
This interface is called ManifestServ like the object it instantiates, but is kept separate from it. This is so the commandline interface can be installed in /usr/bin or other place appropriate for executables, while the definition of the ManifestServ class may be installed in a directory more appropriate for Python library modules (e.g. /usr/lib/python2.4/vendor_packages/...). Usage is as follows:

```
Usage: ManifestServ [-d] [-h|-?] [-s] [-t] [-v] [-f <validation_file_base> ]
   [-o <out_manifest.xml file> ] <manifest.xml file>
where:
  -d: turn on socket debug output (valid when -s also specified)
```

-f <validation_file_base>: give basename for schema and defval files
   Defaults to basename of manifest (name less .xml suffix) when not provided
-h or -?: print this message
-o <out_manifest.xml file>: write resulting XML after defaults and validation processing
-t: save temporary file
   Temp file is /tmp/<manifest_basename>_temp_<pid>
-v: verbose defaults/validation output
-s: start socket server for use by ManifestRead

Please see the ManifestServ __init__() method for more context of these options and what they mean.

12.5.4 *ManifestRead commandline interface*

This commandline interface is intended for shell scripts and programs invoked by the finalizer. It performs the same functions as a ManifestRead object, but on the commandline. Given the socket needed to connect with an active ManifestServ object, and requests, this program displays the results on stdout. The calling shell or program can then pipe the output to a file or iterate on it in a loop for processing.

If a matching node contains a list of data for its value, the list is split and individual list elements are listed on their own output line. If more than one request is given on the commandline or the "-r" option is specified, the nodepath of the request is output alongside its value.

Requests are nodepaths, unless -k is specified in which case requests are keys.

This interface is called ManifestRead like the object it instantiates, but is kept separate from it. This is so the commandline interface can be installed in /usr/bin or other place appropriate for executables, while the definition of the ManifestRead class may be installed in a directory more appropriate for Python library modules (e.g. /usr/lib/python2.4/vendor_packages/...). Usage is as follows:

Usage: ManifestRead [-d] [-h|-?] [-k] [-r] <socket name> <request> [...<request>]
where:
  -d: turn on debug output
  -h or -?: print this message
  -k: specify keys instead of nodepaths
  -r: Always print nodepath next to a value (even when only one nodepath is specified)

Please see the ManifestRead methods for more context of these options and what they mean.

13 Target Instantiation Module

The Target Instantiation module (TI) prepares the target, which has been previously selected for holding an OpenSolaris instance, so that the Transfer Module can appropriately continue with the installation process. The current TI module, developed for other OpenSolaris projects, is also used by the Distribution Constructor to create the build area. The TI module is also used by one of the finalizer scripts DC provides for creating root archives. The TI module needs modifications to it's existing ti_create_target and ti_zfs_list functions. A new function, ti_release_target, will also be added to release the target.

13.1 ti_create_target()

The Distribution Constructor will need modifications to ti_create_target to support the creation of a

basic ufs directory and the creation of a ramdisk boot archive. It will also make use of the creation of the zfs filesystem code.

A new attribute type TI_ATTR_TARGET_TYPE (uint32 array) will be introduced to specify the target type. For the Distribution Constructor the following targets are supported: TI_TARGET_TYPE_DC_UFS, TI_TARGET_TYPE_DC_RAMDISK, and TI_TARGET_TYPE_ZFS.

TI_TARGET_TYPE_DC_UFS will create a basic directory. It will require the attribute TI_ATTR_DC_UFS_DEST(string), where *string* is the name of the directory to be created.

TI_TARGET_TYPE_DC_RAMDISK will create a lofi mounted boot archive. It will require the new attributes TI_ATTR_DC_RAMDISK_SIZE, TI_ATTR_DC_RAMDISK_BOOTARCH_NAME, and TI_ATTR_DC_RAMDISK_DEST. TI_ATTR_DC_RAMDISK_SIZE will accept a uint32 which designates the size of the boot archive to create. TI_ATTR_DC_RAMDISK_BOOTARCH_NAME will accept a string which designates the boot archive file name. And TI_ATTR_DC_RAMDISK_DEST will accept a string which designates the mount point for the bootroot.

TI_TARGET_TYPE_ZFS will create a zfs pool, filesystem and volume. It will require no new attributes.

13.1.1 function declaration

ti_errno_t **ti_create_target**(nv_list_t *, ti_cbf_t)

- nv list describes the target attributes (list of devices, ...)
- ti_cbf_t is callback function type: ti_errno_t (*ti_cbf_t)(nvlist_t *)
- returns result of operation

13.1.2 Name-value pairs describing Target

Name	nv type	Example	Description
TI_ATTR_TARGET_TYPE	uint32_array	TI_TARGET_TYPE_ZFS_FS	Selects target to be created

Table 2. List of attributes describing target type and action

Name	Description
TI_TARGET_TYPE_DC_UFS	DC pkg-image area
TI_TARGET_TYPE_DC_RAMDISK	DC boot archive
TI_TARGET_TYPE_ZFS_POOL	ZFS non-root pool
TI_TARGET_TYPE_ZFS_VOLUME	ZFS volume
TI_TARGET_TYPE_ZFS_FS	ZFS filesystem

Table 3. Available Target Types

Name	Type	Example	Description
TI_ATTR_DC_UFS_DEST	String	/export/tmp	Name of the UFS directory to create

Table 4. List of attributes describing DC UFS directory creation

Name	Type	Example	Description
TI_ATTR_DC_RAMDISK_SIZE	uint32	65536	Ramdisk size in kB
TI_ATTR_DC_RAMDISK_BOOTARCH_NAME	string	boot_archive	Boot archive name
TI_ATTR_DC_RAMDISK_DEST	string	bootroot	Bootroot mount pt.

Table 5. List of attributes describing DC boot archive creation

Name	Type	Example	Description
TI_ATTR_ZFS_POOL_NAME	String	root_pool	Name of pool
TI_ATTR_ZFS_POOL_DEVICE	String	C1t1d0s0	Name of the pool device

Table 6. List of attributes describing ZFS non-root pool creation

Name	Type	Example	Description
TI_ATTR_ZFS_VOL_POOL_NAME	string	zfs_volumes_pool	Name of pool
TI_ATTR_ZFS_VOL_NUM	uint16	2	# of volumes
TI_ATTR_ZFS_VOL_NAMES	string array	vol_swap, vol_dump	Array of volume names
TI_ATTR_ZFS_VOL_MB_SIZES	uint32_array	2048,4096	Array of volume sizes

Table 7. List of attributes describing ZFS volume creation

Name	Type	Example	Description
TI_ATTR_ZFS_FS_POOL_NAME	string	zfs_fs_pool	Name of pool
TI_ATTR_ZFS_FS_NUM	uint16	3	# of data sets
TI_ATTR_ZFS_FS_NAMES	string array	root, usr, opt	Array of data set names
TI_ATTR_ZFS_FS_PROPERTIES	nvlist array	See next table	Data set properties

Table 8. List of attributes describing ZFS file systems

Name	Type	Example	Description
TI_ATTR_ZFS_FS_PROP_NAMES	string array	Compression,quota	Name of properties
TI_ATTR_ZFS_FS_PROP_VALUES	string array	on, 50G	Value of properties

Table 9. List of attributes describing ZFS file system properties.

13.2 ti_zfs_list()

ti_zfs_list() will check to see if the designated dataset exists. This function is used to check for the existence of a particular dataset.

13.2.1 function declaration

ti_errno_t **ti_zfs_list**(nv_list_t *)

- nv list describes the attributes (dataset)
- returns result of operation

13.3 ti_release_target()

ti_release_target() will be a new function to release the target. It will need the same attributes as ti_create_target().

13.3.1 function declaration

ti_errno_t **ti_release_target**(nv_list_t *, ti_cbf_t)

- releases/destroys the target
- nv list describes the target attributes (list of devices, ...)
- ti_cbf_t is callback function type: ti_errno-t (*ti_cbf_t)(nvlist_t *)
- returns result of operation

14 Logging

The DC displays status messages about which program it is executing, and logs all output and errors from the programs. DC logs into 2 files. The simple log only displays the status message from the DC application, and any errors from the programs. The detailed log contains everything in the simple log plus all the output from the image construction. The content of the simple log is also displayed on the window from which the DC application is run. DC's logging feature is implemented using the Python logging module.

The python logger is initialized as soon as the DC application is started. The logger will be named “dc_logger”. Until DC parses the manifest to determine the location of the build area, all log messages will only appear on the window from which the DC application is invoked. Once the location of the build area is determined, warnings and errors will be logged to the simple log, detailed log and the console. All output will be logged to the detailed log. The simple log has the python logging level of INFO. The detail log has the python logging level of DEBUG.

The 2 log files will be stored in <build_area>/logs. Filenames for the simple and detail log are unique cross builds, even if different builds use the same build area. This is useful in case one wants to go back to the previous log files and find out about the details of previous execution attempts. The log file names are computed based on time. The filenames used for each build is displayed on the console so users can easily copy and paste the filename.

The Distribution Constructor calls many programs and scripts to perform it's work. In order to capture the stdout and stderror appropriately in the log files, a new function is defined to execute a given command, and logs it's outputs accordingly.

14.1 exec_cmd_outputs_to_log(cmd, log, stdout_log_level=None, stderr_log_level=None)

This function will execute the given command and send the stdout and stderr to log files.

Arguments:

cmd: The command to execute. The command is expected to have a suitable format for calling Popen with shell=False, ie: the command and its arguments should be in an array.
stdout_log_level: Logging level for the stdout of each command. If not specified, it will default to DEBUG.
stderr_log_level: Logging level for the stderr of each command. If not specified, it will default to ERROR.
Returns:
The return value of the command executed.
Raises:
None

15 Finalizer Script Samples

Some of the steps for creating images are common for most image types. The DC will provide a few scripts for these common operations. Users can use them directly for building their images. These scripts can also be used as starting points for user's custom scripts.
Even though these sample finalizer scripts are provided by DC, they will not called by DC automatically. In order for DC to call these scripts, users need to explicitly specify them in the manifest they use to build their image.

15.1 Populate package image area

This script will populate the package image area in the build area (<build_area>/build_data/pkg_image) with the list of specified IPS packages.
It will perform the following main tasks:

- Read the preferred IPS repository/mirrors, and alternate IPS repositories/mirrors to use from the manifest and set up to use them.
- Read the list of IPS packages to install from the manifest, and install them one by one.
- Read the list of IPS package to remove from the manifest, and uninstall them.
- Read the list of preferred IPS repository/mirrors, and alternate IPS repositories/mirrors to use after installation, and set them in the image.

The Transfer Module, which is used by other Caiman installer projects, also needs to perform many of the operations above.   Section Error: Reference source not found will discuss all the needed modifications to the Transfer Module.

15.2 Root archive construction scripts

Root archive(bootroot) consists of a subset of files from the package image area. For users that want to create bootable images, the following scripts can be useful in constructing the bootroot.

15.2.1 Bootroot initialization script

This scripts "prepares" the bootroot by copying all specified content of the bootroot to the <build_area>/build_data/bootroot directory. This script performs the following actions:

- get the list of files that will be copied from the pkg_image_area to the bootroot from the manifest, and put them in a file.
- use tm_perform_transfer() to copy the list of files into the bootroot directory specifying via arguments that cpio mode is to be used providing the list of files to be copied
- get the list of directories that will be copied from the pkg_image_area to the bootroot from the manifest. Also get the list of directories that need to be excluded from the bootroot, and delete them from the included list.
- loop over the list of directories using tm_perform_transfer() to copy each directory to the bootroot area specifying via arguments that cpio mode is to be used and that a directory is to be recursively copied
- get the list of specified files that need to be excluded from the bootroot, and delete them from the bootroot directory.

15.2.2 Bootroot archive script

This script will actually create the root archive using the prepared content in the bootroot directory. Most users will have at least one finalizer script between the bootroot initialization script and this script to further customize the content of the bootroot.
This script will detect whether the build is done on a SPARC or an x86 system, and create the root archive for the type of system that the build is running on. That means, if you want to create an x86 root archive, you can not do the build on a SPARC system.
This script performs the following actions.

- Based on the content of <build_area>/build_data/bootroot directory, calculates the amount of space needed for the root archive.
- Create a UFS file using the size calculation from the previous step.
- For x86 root archives, copy all the files from the bootroot directory to this root archive file. When all the files are copied, compress the root archive using the algorithm specified in the manifest. If no algorithm is specified, gzip is used.
- For SPARC root archives, copy the files one by one and fiocompress each one of them as they are copied.

16 Transfer Module Modifications

The Transfer module lays out the bits onto the disk. The distribution constructor will add the ability to use IPS to transfer the bits onto the disk. This will include IPS initialization, verification, set authority, unset authority, refresh and retrieval. The TM_perform_transfer function will need to be modified.

There will be two possible CPIO transfer methods, one will cpio the entire contents of the directory to the mount point, the other will cpio files listed in a specified file to the mount point.
A new function to cpio a group of files to the designated mountpoint will need to be created. The function will accept a file which will list all of the files to be cpio'd to the designated mountpoint. This will be specified in the attribute TM_CPIO_LIST_FILE. The mountpoint will be specified in TM_CPIO_DST_MNTPT. TM_ATTR_IMAGE_INFO is optional. This attribute specifies the location of a .image_info file. If the attribute is not specified, none is used.
TM_CPIO_ENTIRE_SKIP_FILE_LIST is an optional attribute to specify a file with a list of files to remove from the destination mountpoint after the cpio is finished.

A new function will be created to cpio an entire directory to the designated mountpoint. This functionwill be equivalent to the old mainline code in the transfer module but will be accessed via the TM_CPIO_ENTIRE attribute.
A new function to perform IPS initialization will need to be created. It will perform the equivalent of the following: pkg image-create -<image type> -a <authority>=<pkg_server>   <ips_mntpt>. TM_IPS_IMAGE_TYPE, TM_IPS_PKG_URL, TM_IPS_PKG_AUTH, and TM_IPS_INIT_MNTPT are the new attributes that are needed by this function. TM_IPS_IMAGE_TYPE is optional and “full” is the default value.

A new function to perform IPS verification will need to be created. It will need to verify that the packages listed in TM_IPS_PKGS are contained in the image's specified authority.
It will also require the attribute TM_IPS_INIT_MNTPT.

A new function to perform the IPS set-authority functionality will be created. It will add an additional authority to the IPS image area. The attributes required are TM_IPS_INIT_MNTPT, TM_IPS_ALT_AUTH, and TM_IPS_ALT_URL.
A new function to perform the IPS unset-authority functionality will be created. It will remove the designated authority from the IPS image area. The attributes required are TM_IPS_INIT_MNTPT and TM_IPS_ALT_AUTH.

A new function to perform the IPS refresh functionality will be created. It will refresh the IPS image area. The attribute required is TM_IPS_INIT_MNTPT.

A new function to perform IPS retrieval will need to be created. The function will retrieve the packages listed in TM_IPS_PKGS. The required attributes are TM_IPS_INIT_MNTPT and TM_IPS_PKGS.

16.1 TM_perform_transfer()

The nvlist_t that is passed into TM_perform_transfer will need to be modified to be an array of attributes. New attributes to be added are TM_PERFORM_CPIO and TM_PERFORM_IPS. Each of these attributes will need additional attributes to be passed to provide more information. The function will need to be modified to add code to perform an IPS init, verification, set authority, unset authority, refresh, and retrieval given the designated attributes. A return code indicating success or failure and some information as to what functionality failed is necessary.

16.1.1.1 data structures

```
typedef enum {
	TM_E_SUCCESS = 0,		/* command succeeded */
	TM_E_INVALID_TRANSFER_TYPE_ATTR, /* transfer type attr invalid */
	TM_E_INVALID_CPIO_ACT_ATTR,	/* cpio transfer type attr invalid */
	TM_E_CPIO_ENTIRE_FAILED,	/* cpio of entire dir failed */
	TM_E_INVALID_CPIO_FILELIST_ATTR, /* cpio filelist attr invalid */
	TM_E_CPIO_LIST_FAILED,		/* cpio of file list failed */
	TM_E_INVALID_IPS_ACT_ATTR, /* ips transfer type attr invalid */
	TM_E_INVALID_IPS_SERVER_ATTR, /* ips server attribute invalid */
	TM_E_INVALID_IPS_MNTPT_ATTR, /* ips init mountpoint invalid */
	TM_E_IPS_INIT_FAILED,		/* ips initialization failed */
	TM_E_IPS_CONTENTS_VERIFY_FAILED,	/* ips verification failed */
	TM_E_IPS_RETRIEVE_FAILED,	/* ips retrieval failed */
	TM_E_ABORT_FAILED,		/* abort failed */
	TM_E_REP_FAILED,		/* progress report failed */
	TM_E_IPS_PKG_MISSING,		/* ips package not found in repository */
	TM_E_IPS_REFRESH_FAILED,	/* ips refresh failed */
	TM_E_IPS_SET_AUTH_FAILED,	/* ips set-auth failed */
	TM_E_IPS_UNSET_AUTH_FAILED,	/* ips unset-autho failed */
	TM_E_PYTHON_ERROR		/* general python error */
} tm_errno_t;
```

tm_errno_t TM_**perform_transfer**(nv_list_t *, ti_cbf_t)

- nv list describes the target attributes (list of devices, ...)
- ti_cbf_t is callback function type: ti_errno_t (*ti_cbf_t)(nvlist_t *)
- returns result of operation

16.2 Name – value Pairs describing Target

Name	nv type	Example	Description
TM_ATTR_MECHANISM	uint32	TM_PERFORM_CPIO	Selects type of transfer

Table 2. List of attributes describing transfer type

Name	Description
TM_PERFORM_CPIO	Perform cpio copy
TM_PERFORM_IPS	Perform IPS package retrieval

Table 3. Available transfer Types

Name	Type	Example	Description
TM_CPIO_ACTION	int32	TM_CPIO_ENTIRE	Selects which cpio action to perform

Table 4. List of attributes describing CPIO transfer

Name	Description
TM_CPIO_ENTIRE	cpio the entire distribution
TM_CPIO_LIST	cpio only specified files

Table 5. Available CPIO transfer actions

Name	Type	Example	Description
TM_CPIO_DEST_MNTPT	String	/export/tmp	Name of the mount point to cpio to
TM_CPIO_SRC_MNTPT	String	/export/proto	Name of the mount point to cpio from
TM_ATTR_IMAGE_INFO	string	/.cdrom/.image_info	Location of the .image_info file
TM_CPIO_ENTIRE_SKIP_FILE_LIST	File	/export/skip_files	File with list of files to remove when the cpio has finished.

Table 6. List of attributes describing cpio entire transfer

Name	Type	Example	Description
TM_CPIO_DEST_MNTPT	String	/export/tmp	Name of the mount point to cpio to
TM_CPIO_LIST_FILE	String	/tmp/cpiolist	File containing a list of files to cpio
TM_ATTR_IMAGE_INFO	String	/.cdrom/.image_info	Location of the .image_info file

Table 7. List of attributes describing cpio list transfer

Name	Type	Example	Description
TM_IPS_ACTION	Int32	TM_IPS_INIT	Selects which ips action to perform

Table 8. List of attributes describing IPS transfer

Name	Description
TM_IPS_INIT	Perform IPS initialization
TM_IPS_CONTENTS_VERIFY	Verify the IPS packages are in the repository
TM_IPS_RETRIEVE	Perform IPS package retrieval
TM_IPS_SET_AUTH	Add an authority to the IPS image
TM_IPS_UNSET_AUTH	Remove authority from the IPS image
TM_IPS_REFRESH	Refresh the IPS image

Table 9. Available IPS transfer actions

Name	Type	Example	Description
TM_IPS_PKG_URL	string	http://opensolaris.sun.org	IPS respository location
TM_IPS_PKG_AUTH	String	Opensolaris.org	Authority
TM_IPS_INIT_MNTPT	string	/tmp/proto	Directory to initialize

Table 10. List of attributes describing IPS initialization

Name	Type	Example	Description
TM_IPS_PKGS	string	/distro/pkgs.txt	File with list of packages
TM_IPS_INIT_MNTPT	String	/export/home/image	IPS image area

Table 11. List of attributes describing IPS verification

Name	Type	Example	Description
TM_IPS_PKGS	string	/distro/pkgs.txt	File with list of IPS packages
TM_IPS_INIT_MNTPT	String	/export/image	IPS image area

Table 12. List of attributes describing IPS retrieval

Name	Type	Example	Description
TM_IPS_INIT_MNTPT	String	/export/image	IPS image area
TM_IPS_ALT_AUTH	String	Example.com	Authority
TM_IPS_ALT_URL	String	Http://mysolaris:10	IPS repository

Table 13. List of attributes describing IPS set-auth

Name	Type	Example	Description
TM_IPS_INIT_MNTPT	String	/export/image	IPS image area
TM_IPS_ALT_AUTH	String	http://mysolaris:10	IPS repository

Table 14. List of attributes describing IPS unset-auth

Name	Type	Example	Description
TM_IPS_INIT_MNTPT	String	/export/image	IPS image area

Table 15. List of attributes describing IPS refresh

16.3 TM_abort_transfer()

TM_abort_transfer needs no modifications.

17 References

- Distribution Constructor web page: http://opensolaris.org/os/project/caiman/Constructor/
- PSARC/2007/039 – Caiman: New Solaris Installation Experience
- PSARC/2008/190 – Image Packaging System
- OpenSolaris Distribution Constructor Guide: http://dlc.sun.com/osol/docs/content/dev/DistroConst/
- distro_const(1M) man page: http://src.opensolaris.org/source/xref/caiman/slim_source/usr/src/man/distro_const.1m.txt
- Distribution Constructor Test Plan: http://ontestreview.central.sun.com/wiki/index.php/Distro_Constructor_tp