Build Environment HLD for Colibri project

By Yuriy V. Umanets <yuriy.umanets@clusterstor.com>

Date: 2010/02/19
Revision: 1.0

[Text in square brackets and with a light-green background is a commentary explaining the structure of a design document.]

This document presents a high level design (HLD) of a build environment for an upcoming T1 Colibri project. The main purposes of this document are: (i) to be inspected by T1 architects and peer designers to ascertain that high level design is aligned with T1 architecture and other designs, and contains no defects, (ii) to be a source of material for Active Reviews of Intermediate Design (ARID) and detailed level design (DLD) of the same component, (iii) to serve as a design reference document.

The intended audience of this document consists of Colibri customers, architects, designers and developers.

0. Introduction

[This section succinctly introduces the subject matter of the design. 1--2 paragraphs.]

Colibri project requires sophisticated build system with many features. It also requires logical directory structure, so that modifications are done quickly and painlessly. This document aims to describes such a build environment.

1. Definitions

[Definitions of terms and concepts used by the design go here. The definitions must be as precise as possible. References to the C2 Glossary are permitted and encouraged. Agreed upon terminology should be incorporated in the glossary.]
build system - Autotools, Cmake, Scons, etc., that is, set of software packages, which may work together to provide the means of building complex software packages;\

build env - build system plus project policies, such as directory structures, practices used for adding new targets, etc.

autotools - GNU Build system, that is: GNU Automake and GNU Autoconf.

automake - GNU Automake is a package, containing set of utilities for generating make files templates from special make target definitions, called Makefile.am. It is closely tied to autoconf (see below) and usually used along with it as part of bigger processing pipeline. As a result of automake work, Makefile.in files are generated.

autoconf - GNU Autoconf is a package, containing set of utilities for configuring projects and generating make files from special templates, generated by automake (see above) and called Makefile.in. As a result of its work, special configure script is created and used for checking system environment. When checking is done, make files are generated, taking into account site environment, build tools and build options specifics, found in configure time.

libtool - GNU Libtool is a package, providing portable way for building static or dynamically linked libraries by hiding platform specifics and linker/compiler options, required for building a library, in a huge, presumably portable shell script. It is usually used along with automake and autoconf.

scons - modern, portable build system, alternative to autoconf/automake, though having all the features of them and even more. It is implemented in python and suggests, that build scripts are also written in python. This provides big deal of flexibility and potential as one can use power of python and whole bunch of existing libraries.

cmake - one more alternative build system used for big projects. Its main difference from others is that, it generates build scripts/files, native to each platform it is used on from some very simple script, describing project checks, targets, etc. Native scripts for each platform is: makefiles for unices, MS VisualStudio project for Windows and Eclipse project for java applications.

2. Requirements

[This section enumerates requirements collected and reviewed at the Requirements Analysis (RA) and Requirements Inspection (RI) phases of development. References to the appropriate RA and RI documents should go here. In addition this section lists architecture level requirements for the component from the Summary requirements table and appropriate architecture documentation.]
[**req.build-env.traditional]** - All the traditional features of build systems, such as using timestamps optimization, build correctness, should be supported.

[req.build-env.portability] - The same build system and source tree should be used for building Colibri on all supported platforms.

[req.build-env.user-kernel] - The source code tree contains user space code and targets (libraries, binaries) as well as kernel space code (kernel modules).

[req.build-env.configure-stage] - Configure stage should be present to check various system specifics, libraries, functions, headers.

[req.build-env.configure-options] - Configure stage should support passing various configure options, such paths to other software packages to use, debug on/off, etc.

[req.build-env.build-stage] - Build stage should be powerful enough to be able to do non-traditional build tasks, such as generating code from templates, execute shell commands, etc.

[req.build-env.simultaneous-builds] - Simultaneous builds from the same or different nodes should be supported. Different architectures may be built simultaneously.

[req.build-env.build-dir] - All object files, binaries and libraries, should be placed out of the source tree.

[req.build-env.debug] - Debug on/off targets are supported and placed in different directories.

[req.build-env.performance] - Complex project structure is supported and handled in a sophisticated way to provide as fast builds as possible.

[req.build-env.testing] - Testing framework is built along with the project software.

[req.build-env.git-repo] - Git repository is used for storing the project source code and build system files.

[req.build-env.git-hooks] - Git commit hooks are set in the repository to mail code changes to the peers.

3. Functional specification

[This section defines a functional structure of the designed component: the decomposition showing *what* the component does to address the requirements.]

To meet the requirements, three main parts of the build environment should be implemented:

1. *Configuration part.* It should make sure, that required portability level is reached. It also should implement build options, such as paths to the software packages we use, debug on/off, etc;
2. *Generation part.* Objects and binaries generation part of the build process should be implemented with focus on portability, performance and correctness. Both, configuring and generation parts should support the ability to generate binaries in a directory away of source code. This enables simultaneous builds as well as minimizes mess in source tree after the project is built;
3. *Repository part.* All the source code should be placed in number of git repositories with all the scripts, allowing comfortable development.

3.1. Configuration part

Here is the list of what should be checked by configure part:

- Architecture, operating system, endianes - this may be required for small workarounds based on this information;
- All the tools required for building the project - compiler, linker, man generator, etc;
- Header files we use in the project. They may be different or placed in different locations;
- Libraries we link against in the project. Some functions may be placed in different libraries on

different platforms. For example, pthread API functions are located in libpthread on Linux and libc on Mac OS X systems;
- Functions, which we know are not standard or may be different on some platforms (direct I/O);
- Optional things, such as debug on/off or things like whether to build or not server or client, should be moved to build/configuring options in a way like this: debug=1;
- Any software packages we need to build the project, such as db4 or Linux kernel, should be specified in the way like this: with-gss=gss-path with-linux=linux-kernel-path.

The following configuring options form configure part functionality:

Name	Description
debug=on/off	Specified whether to build with or without debug information for the objects and binaries. Off by default.
with-db4=db4-path	Db4 location we want to use for the metadata server. May be skipped, in this case our own db4 is used.
with-linux=linux-path	Linux kernel location we want to use for kernel components.
with-gss=gss-path	Gss libraries location we want to use for security.
build-dir=build-path	Specifies the location of where object files and binaries should be generated.

3.2. Generation part

Here is the list if build part functionality:

1. Options provided by configure part are taken into account, including paths, debug and optimization options;
2. Support for simultaneous builds and generating binaries out of the source tree;
3. Support for features like generating source code from templates (db4 fol records).

3.4. Repository part

The following functionality is provided:
1. Source code is stored in a repository, supporting branches and commit hooks;
2. Commit hooks are set to mail repository changes to the peers. Commit emails contain standard diff of the changes maid.

4. Logical specification

[This section defines a logical structure of the designed component: the decomposition showing *how* the functional specification is met. Subcomponents and diagrams of their interrelations should go in this section.]

4.1. Configuration part

Configuration part, in whatever build system it is implemented, check the following system specifics:
- Platform check - part of standard check in Autotools and Cmake. Very easy to figure out in Scons;
- Build tools, such as compiler, linker, etc., are usualy the part of standard configure checks for all build systems;

- Header files check - one line macro/function in all build systems;
- Libraries checks - one line macro/function in all build systems;
- Not standard functions - one line macro/function in all build systems;
- Configure options - handled differently in all build systems but still is very easy to talk much about. DLD will show examples of all the checks including configure options checks.

4.2. Generation part

Generation part is very simplistic for all of the build systems described. One just specifies what targets needed: binaries, libraries, etc. The only exception is that, we may possibly need handling for not standard source files, such as fol template, to generate fol tables for db4.

4.3. Repository part

So far there are three repositories for storing different logical parts of the Colibri source tree:

Name	Description
colibri	git clone ssh://gitosis@git.clusterstor.com/colibri.git Root repository holding the following: ■ colibri-core - colibri repository; ■ colibri-db4 - modified db4 repository; ■ colibri-get - bash script for fetching both repositories.
colibri-core	git clone ssh://gitosis@git.clusterstor.com/colibri-core.git Repository for storing colibri-core (Colibri itself) source code.
colibri-db4	git clone ssh://gitosis@git.clusterstor.com/colibri-db4.git Modified version of db4, which we can build against. It will contain all the fixes we need: direct IO, dirty pages flush fixes, etc.

As one can see, colibri-core and colibri-db4 are separate repositories. There is no some logical explanation why we need them separately now as there are no requirements dictating this structure. We surely can have such a requirements in future and more flexible repository structure allows to adopt quickly. Additionally, there are not many drawbacks in current structure to not do so.

While colibri-db4 repository is having completely the same structure as standard db4 distribution (we do not want to multiply differences, so that we could switch to new version quickly), colibri-core, on the other hands, having the structure, discussed bellow.

- **doc** - documentation such as installation guide, APIs description;
- **man** - standard manual pages for binaries, config files (if any), etc;
- **src** - source code;
  - **addb** - addb module;
  - **ctdb** - ctdb module;
  - **fol** - fol module;
  - **nrs** - nrs module;
  - **sns** - sns module;

    - **net** - networking code;
    - **lib** - common code for all the modules;
    - ...
- **include** - all colibri include files;
- **patches** - db4 patches;
- **tests** - testing framework.

Every subdirectory, which contains something to build (code, man pages, etc), should have own build script. This build script builds only thing related to its directory. This is worth to mention, as some build systems allow to do everything from the project root directory. This would make root build script huge and even the best build system in the world will not save it from becoming mess very quickly.

It is important to configure the project before building. This allows build system to adopt to different versions of libraries installed, find compiler and linker paths, etc., giving the ability to be as portable as possible.

4.4. Conformance

[For every requirement in the Requirements section, this sub-section explicitly describes how the requirement is discharged by the design. This section is part of a requirements tracking mechanism, so it should be formatted in some way suitable for (semi-)automatic processing.]

[req.build-env.traditional] - Covered with configuration part of the build environment.

[req.build-env.portability] - Portability requirement is covered by build system used for the build environment. Some of them are more better in this: scons uses python or generates native IDE projects, cmake generates native IDE projects on Windows and uses make on Unix. Autotools is less convenient as it has more difficulties to build things on Windows.

[req.build-env.user-kernel] - Configure stage allows to specify linux kernel location. This theoretically allows to build kernel components in same tree with user space components, according kernel manuals on how to build kernel components out of kernel source tree.

Issue is that, kernel is built using make and its out-of-tree building is based on make. One needs to create specially formatted make file to be able to build kernel modules. This means that scons cannot be used for building kernel modules unless somebody adds scons support to linux kernel. Or alternatively we can call make from scons.

That said, to cover this requirement we need to use Autotools (make based build system) and this contradicts to other requirements (portability). This may end up with attempt to use two build systems in parallel, which leads to more complexity and bugs.

[req.build-env.configure-stage] - All build systems have powerful configure functionality, in which they check all the system specifics, such as platform, endianess, headers, etc.

[req.build-env.configure-options] - All build systems, which could be used by Colibri build environment, may easily pass configure options down to the generation part.

[req.build-env.build-stage] - All build systems have build stage, which is usually follows after configure stage and builds objects, using configure options and findings.

[req.build-env.simultaneous-builds] - Most of build systems can do simultaneous builds (parallel builds in Autotools manual). Still, scons is doing this the way exactly we need, that is, generates build tree in specified location without needs to do something more by hands. Latter is required for Autotools and Cmake. They imply, that one does this: mkdir build; cd build; configure ... && make.

[req.build-env.build-dir] - Scons build system allows easily to specify the directory all the binaries should be generated in. Others do not allow to do it simple-portable way.

[req.build-env.debug] - Just one of options. May be handled easily by all build systems.

[req.build-env.performance] - Scons provides slightly more flexibility here due to rich set of "deciders" - functions to check if source file has changed since last build. It also may be much faster than autotools because of caches and different way of building the sources (no need to chdir to every single directory all the time).

[req.build-env.testing] - We just create tests directory, place tests code there and build it all along with the rest of the package.

4.5. Dependencies

[This sub-section enumerates other system and external components the component depends on. For every dependency a type of the dependency (uses, generalizes, *etc.*) must be specified together with the particular properties (requirements, invariants) the design depends upon. This section is part of a requirements tracking mechanism.]

- configure-stage
    - scons
        - python
    - autotools
        - bash
            - cygwin (Windows)
    - cmake
        - bash
            - cygwin (Windows)
        - cmake interpreter
- build-stage
    - scons
        - python
        - make (for linux kernel modules)
        - gcc, linker
    - autotools

        - make
        - gcc, linker
    - cmake
        - native build tool (eclipse, MS Visual studio, etc)
- git
    - server side hooks

4.6. Security model

[The security model, if any, is described here.]

Security model may be applied to the repository used for the project. The are two options available:

1. gitosis model is used. This is when each repository user needs to send his/her public ssh key to the system administrator to be able to access the repositories;
2. github model is used. This is when the repository is stored on github service and its security model (also based on ssh keys) is used.

We think, that using first model is more appropriate here, as it provides all kinds of safety for the code as we fully control the repository. Source code is placed physically on the server owned by company. And second option is less convenient as makes code stealing theoretically possible as all is placed on servers not controlled by our company and accessible through the web interface. It also provides some kind of knowledge of what is done and what is going to be done about the project to competitors, which is definitely not an option.

4.7. Refinement

[This sub-section enumerates design level requirements introduces by the design. These requirements are used as input requirements for the detailed level design of the component. This sub-section is part of a requirements tracking mechanism.]

4.7.1. Scons specifics

- Scons scripts run into two phases:
    1. *Preparation.* All the code in script is executed. All targets like Program, Library or Object are added to special construction list;
    2. *Construction.* Construction list, formed in previous stage, is processed to build all the targets specified. Everything that builds with custom builder, will be also built in construction phase. This means, that custom builders are better not used for generating source code from templates, because generation will happen in construction phase, where the source code has to be already generated.
- There is possibility to define custom Builder (builds non-standard input files) and Scanner (parses non-standard input files).

4.7.2. Linking against not installed libraries

One more thing to work out deeply in the DLD is linking against not installed shared libraries. We may potentially have this need to link against our custom db-4 version. Issue is that, linking against such libraries means that they cannot be found by dynamic linker and as such, cannot be used without additional hands work like setting up LD_LIBRARY_PATH. Worth to say, that most of build system rely and recommend to link against installed libraries.

This all raises couple of rather important questions, which should be discussed in the DLD:

- Do we really need to link against not-installed db-4 libraries or we better install them?
- If we do install them, should we rename them to not conflict with already installed libraries in the system or should we maintain our custom db-4 APIs the way that it is still usable for the rest of system tools?
- If we do not install our custom db-4, is it good style to link against it statically?

4.7.3. Building kernel modules

There is an issue with building kernel modules using scons. As kernel is make based, one needs to generate specially formatted makefiles and call make from scons script. This is not looking very well. We need to think on how to solve this.

There are three ways of solving this:

1. Generate makefiles and call make from scons;
2. Reject scons completely and use Autotools (contradicts to portability use cases);
3. Add scons support to linux kernel (lots of effort and out of our main focus).

5. State

[This section describes the additions or modifications to the system state (persistent, volatile) introduced by the component. As much of component behavior from the logical specification should be described as state machines as possible. The following sub-sections are repeated for every state machine.]

5.1. States, events, transitions

[This sub-section enumerates state machine states, input and output events and state transitions incurred by the events with a table or diagram of possible state transitions. UML state diagrams can be used here.]

Events:

- debug on/off is specified;
- with-db4 option is specified;
- architecture is detected;
- build dir is specified;
- build failure.

States:

- configuring;
- generation.

5.2. State invariants

[This sub-section describes relations between parts of the state invariant through the state modifications.]

Whole the build environment changes its state from *configuring* to *generating*. In scons this is done seamlessly, without user attention. Still these two states are separated in time.

5.3. Concurrency control

[This sub-section describes what forms of concurrent access are possible and what forms on concurrency control (locking, queuing, *etc*.) are used to maintain consistency.]

[concur.simultaneous-build.different-nodes] - Separate build directories are used for maintaining consistency. Dir names are formed using node name and architecture name. In correctly configured network, this ensures correct simultaneous build;

[concur.simultaneous-build.same-node] - Separate build directories are used for maintaining consistency. Directory names are specified by user, running the build command.

6. Use cases

[This section describes how the component interacts with rest of the system and with the outside world.]

6.1. Scenarios

[This sub-section enumerates important use cases (to be later used as seed scenarios for ARID) and describes them in terms of logical specification.]

The following use cases are important to consider to define main features of the build environment.

Scenario	use-case.simultaneous-build
Relevant quality attributes	scalability
Stimulus	simultaneous build from different nodes with different architecture (no build-dir specified)
Stimulus source	user
Environment	generate-part
Artifact	build script creates a build dir name, using architecture information gathered in configure stage and instructs build system to use it as "variant_dir"
Response	binaries and objects are placed in the build dir, out of the source code dir, this makes it possible to build in parallel as different nodes used different build dirs
Response measure	build dir contains replicated project directory structure with all the objects and binaries built (sources also may be there optionally for better portability)
Questions and issues	-

Scenario	use-case.build-dir-specified
Relevant quality attributes	flexibility

Stimulus	generate binaries in a dir specified by user, which is located out of source code
Stimulus source	user
Environment	generate-part
Artifact	the build dir is specified in configure time
Response	generate-part makes sure, that binaries and objects are placed in the build dir. May be used for building from different nodes, given they specify different build directories.
Response measure	build dir contains replicated project directory structure with all the objects and binaries built (sources also may be there optionally for better portability)
Questions and issues	-

Scenario	use-case.windows-build
Relevant quality attributes	portability
Stimulus	project is built in windows
Stimulus source	user
Environment	generate-part
Artifact	build is run in Windows
Response	build-env is able to find compiler and linker (both use correct options), build scripts them selves can run in Windows, build is done without issues
Response measure	Windows (PE) executables are generated
Questions and issues	This implies existing of build system package and also build script interpretor (python or bash)

Scenario	use-case.db4-location-specified
Relevant quality attributes	usability
Stimulus	db4 location is specified as configuration option
Stimulus source	user
Environment	configure-part, generate-part
Artifact	db4 location needs to be checked and passed to the generation-part
Response	configure-part checks if db4 location contains correctly built db4 of specific version with all patches applied and passes it to generate-pare. Generation part uses db4 location for includes and libraries search paths. It also generates fol record source code using special awk script, located at db4 location.
Response measure	binaries are successfully linked against db4 libraries, which may be checked with standard nm (for statically linked libraries) and ldd (for dynamically linked libraries) tools.
Questions and issues	This scenario is used for the case build against db4, which is not part of our distribution.

Scenario	use-case.db4-location-missed
Relevant quality attributes	usability
Stimulus	db4 location is not specified as configuration option
Stimulus source	user
Environment	configure-part, generate-part
Artifact	db4 is not specified. This means we want to build our own db4 and use it for building

	T1
Response	if db4 location is not specified, we conclude, that user wants to use our distribution db4. We build db4 using configure options we need for our project and use its location for all the checks, etc., according to *[use-case.db4-location-specified]*
Response measure	binaries are successfully linked against db4 libraries, which may be checked with standard nm (for statically linked libraries) and ldd (for dynamically linked libraries) tools.
Questions and issues	-

Scenario	use-case.debug-option-specified
Relevant quality attributes	usability
Stimulus	debug on/off is specified as configuration option
Stimulus source	user
Environment	configure-stage, generate-stage
Artifact	debug flags are specified
Response	configure-stage passes debug on/off flag down to the generation-stage
Response measure	binaries contain debug information (debug=on case), which may be checked with binutils
Questions and issues	-

Scenario	use-case.kernel-module-build
Relevant quality attributes	usability
Stimulus	with-linux configure option is specified
Stimulus source	user
Environment	configure-stage, generate-stage
Artifact	linux kernel location is specified, need to build kernel modules
Response	configure-stage passes kernel path down to the generation-stage
Response measure	kenel is make based software, which means we need create special make file on flight and call make from scons or we need to use autotools
Questions and issues	this use case shows that we either need to generate makefiles from scons in order to build kernel modules or reject using scons at all (contradicts to portability related use cases)

One of issues that we solve in this HLD is choosing build system, which is in line with our use cases more than others.

We checked three build systems:

1. Autotools;
2. Scons;
3. CMake.

The following table helps to see, how specific build system conforms to use cases described above.

-	Autotools	Scons	CMake	Comments
use-case.simultaneous-build	1	1	1	All allow to start several simultaneous builds
use-case.build-dir-specified	0.8	1	0.8	It is only in scons one may specify which dir is used for building binaries. Others require creating build dir by hands and running configure and build from that directory.
use-case.windows-build	0.6	1	1	Bash, libtool, etc., in Autotools is definitely less portable than python in scons and native projects in cmake. Autotools in Windows requires Cygwin.
use-case.db4-location-specified	1	1	1	All allow to specify and check for any options.
use-case.db4-location-missed	1	1	1	All allow to specify and check for any options.
use-case.debug-option-specified	1	1	1	All allow to specify and check for any options.
use-case.kernel-module-build	1	0.5	0.9	Autotools and Cmake are better in this as they both are make based in Unix environment.

6.2. Failures

[This sub-section defines relevant failures and reaction to them. Invariants maintained across the failures must be clearly stated. Reaction to Byzantine failures (*i.e.*, failures where a compromised component acts to invalidate system integrity) is described here.]

Scenario	failure-case.invalid-db4-location
Description	Configure part detects, that db4 location, specified by user input, does not contain correctly build db4 libraries or libraries version is wrong.
Reaction	Raise an error message and stop the current stage (configure-part)

Scenario	failure-case.invalid-architecture
Description	Not supported architecture/platform detected
Reaction	Raise an error message and stop the current stage (configure-part)

Scenario	failure-case.invalid-env

Description	Build environment issues, such as wrong library or header, compiler, etc
Reaction	Raise an error message and stop the current stage (configure-part)

The rest of failures are handled automatically by the build system (autotools or scons).

7. Analysis

7.1. Scalability

[This sub-section describes how the component reacts to the variation in input and configuration parameters: number of nodes, threads, requests, locks, utilization of resources (processor cycles, network and storage bandwidth, caches), *etc*. Configuration and work-load parameters affecting component behavior must be specified here.]

7.1.1. Simultaneous builds

Theoretically, for the build system, it is not important how many nodes try to build the system in parallel. This is rather matter of file-system in use and its abilities to handle such a load efficiently.

7.1.2. Distributed builds

This is an interesting possibility, which may be supported using distc. Still we are not interested in it, as T1 and whole Colibri should not be as huge to build hours. We need such a good build environment, which allows to build quickly.

7.2. Rationale

[This sub-section describes why particular design was selected; what alternatives (alternative designs and variations of the design) were considered and rejected.]

7.2.1. Autotools

Autotools build system was rejected as one, that is not completely in line with the project requirements. The main downsides from the point of view of this project are the following:

- Portability of autotools is not good enough even between Unices. Situation with Windows is even worse;
- Simultaneous builds approach, suggested by autotools is less convenient, as it implies creating build dir and running build from there.

7.2.2. CMake

One more alternative build system, which was rejected. Here is the rationale:

- Building out of source uses the same approach as autotools, which we have already rejected as less convenient than one used in scons;
- It depends on make on Unix systems;
- CMake language is not as rich as python is. This language needs to be learned and python is known to large number of people.

Of course CMake has its unique nice features. One of them is build dashboard, accessible through the web. It shows builds status, errors, which nodes built what, etc. Truth is, we really not sure that we need this and we're definitely not ready to sacrifice the other needs for this one.

8. Deployment

8.1. Installation

[How the component is delivered and installed.]

There are two types of installation possible:

1. Using installed scons;
2. Distribute scons with our package.

Using installed scons:

1. Clone the repository as it was shown above;
2. Make sure that python and scons are installed;
3. Run the build system from the project root directory.

Distribute scons:

1. Scons may be installed from our package;
2. Scons may be executed without installing.

8.2. Using build environment

Building colibri:
```
cd ./colibri
scons [target]
```

Install colibri:
```
cd ./colibri
scons install
```

Cleanup colibri:
```
cd ./colibri
scons clean
```

Make a distribution archive:
```
cd ./colibri
scons dist
```

8.3. Troubleshooting

In some cases the package cannot be built due to environment issues, bugs in build script, etc. Or build system behaves the way, which is not expected (misses some checks, finds wrong libraries or headers). This chapter explains how to figure out what is wrong.

Scons build system, which we have already agreed on using, provides rich set of troubleshooting mechanisms.

They are the following:

8.3.1. Rebuilding decisions

We may need to know why some target gets rebuilt. To figure this out, we can use the following build command:

```
scons --debug=explain
```

The output will be like this:

```
scons: building `build/x86_64.victim.localdomain/src/addb/addb.os' because it
doesn't exist
gcc -o build/x86_64.victim.localdomain/src/addb/addb.os -c -O2 -fPIC -
Iinclude -I. src/addb/addb.c
```

8.3.2. Environment dumping

Most of things in scons are done on behalf of some environment instance. These confgure part checks as well as generating targets. Environments contain all the paths to build tools (compiler, linker), paths to libraries and headers, optimization flags, custom definitions derived from upper level build scripts, etc. As such it becomes very important to know what does particular environment contain in case of errors.

This may be done like this following. In case of error occurred during building a target, build script may dump environment in the way showed below:

```
if error:
      print env.Dump()
```

An result of this command will look like this following:

```
[umka@victim colibri-core]$ scons
scons: Reading SConscript files ...
Building at: build/x86_64.victim.localdomain
{ 'AR': 'ar',
  'ARCOM': '$AR $ARFLAGS $TARGET $SOURCES',
  'ARFLAGS': ['rc'],
  'AS': 'as',
  'ASCOM': '$AS $ASFLAGS -o $TARGET $SOURCES',
  'ASFLAGS': [],
  'ASPPCOM': '$CC $ASPPFLAGS $CPPFLAGS $_CPPDEFFLAGS $_CPPINCFLAGS -c -o
$TARGET $SOURCES',
  'ASPPFLAGS': '$ASFLAGS',
  'BUILDDIR': 'build/x86_64.victim.localdomain',
  ...
```

What is shown is only small part of the dump.

8.3.3. Dependencies tracking

Scons uses either implicit dependencies collecting mechanism (when you point out to use some headers directory) or explicit collecting mechanism (when you clearly state in the script, that your target depends on particular file or directory).

In some cases it is useful to see what dependencies are for particular target. The way to make scons show this is the following:

```
scons --tree={all,status,derived,prune}
```

The value of --tree option may be combined from several options in way like this:

```
scons --tree=status,derived
```

An result of --tree=all command looks like this following:

```
scons: done reading SConscript files.
scons: Building targets ...
scons: `.' is up to date.
+-.
  +-.sconf_temp
  +-SConstruct
  +-build
  | +-build/x86_64.victim.localdomain
  |   +-build/x86_64.victim.localdomain/doc
  |   | +-doc/SConscript
  |   +-build/x86_64.victim.localdomain/man
  |   | +-man/SConscript
  |   +-build/x86_64.victim.localdomain/src
  |   | +-build/x86_64.victim.localdomain/src/addb
  |   | | +-src/addb/SConscript
  |   | | +-src/addb/addb.c
  |   | | +-build/x86_64.victim.localdomain/src/addb/addb.os
  |   | | | +-src/addb/addb.c
  |   | | | +-include/colibri/colibri.h
  |   | | | +-/usr/bin/gcc

  ...

  |   | +-build/x86_64.victim.localdomain/src/fol
  |   | | +-src/fol/SConscript
  |   | | +-src/fol/colibri_fol.c
  |   | | +-src/fol/colibri_fol.h
  |   | | +-build/x86_64.victim.localdomain/src/fol/colibri_fol.os
  |   | | | +-src/fol/colibri_fol.c
  |   | | | +-src/fol/colibri_fol.h
  |   | | | +-/home/umka/Work/colibri/colibri-db4/build_unix/db.h
  |   | | | +-/home/umka/Work/colibri/colibri-db4/build_unix/db_config.h
  |   | | | +-/home/umka/Work/colibri/colibri-db4/build_unix/db_int.h
  |   | | | +-/home/umka/Work/colibri/colibri-db4/dbinc/db_swap.h
  |   | | | +-src/fol/dbtypes.h

  ...
```

```
 +-config.log
  +-doc
  | +-doc/SConscript
  +-include
  | +-include/colibri
  |   +-include/colibri/colibri.h
  +-man
  | +-man/SConscript
  +-src
  | +-src/addb
  | | +-src/addb/SConscript
  | | +-src/addb/addb.c
  | +-src/ctdb
  | | +-src/ctdb/SConscript
  | | +-src/ctdb/ctdb.c
  | +-src/fol
  | | +-src/fol/SConscript
  | | +-src/fol/colibri_fol.c
  | | +-src/fol/colibri_fol.h
  | | +-src/fol/colibri_fol_print.c
  | | +-src/fol/colibri_fol_template.c
  | | +-src/fol/dbtypes.h
```

Note that both, build tree and source tree dependencies are shown.

8.3.4. Command lines constructing

One more useful debugging ability is to control how do command lines, used for building targets, are constructed by buil system. Scons provides the following tool:

```
scons -Q --debug=presub
scons: done reading SConscript files.
scons: Building targets ...
Building build/x86_64.victim.localdomain/src/addb/addb.os with action:
  $SHCC -o $TARGET -c $SHCFLAGS $SHCCFLAGS $_CCCOMCOM $SOURCES
gcc -o build/x86_64.victim.localdomain/src/addb/addb.os -c -O2 -fPIC -
Iinclude -I. src/addb/addb.c
```

8.3.5. Find libraries

```
scons --debug=findlibs
gcc -o build/x86_64.victim.localdomain/src/fol/colibri_fol_template.os -c -O2
-fPIC -Iinclude -I. -Ibuild/x86_64.victim.localdomain/src/fol -Isrc/fol -
I/home/umka/Work/colibri/colibri-db4 -I/home/umka/Work/colibri/colibri-
db4/build_unix src/fol/colibri_fol_template.c
  findlibs: looking for 'libdb-4.8.a' in '/home/umka/Work/colibri/colibri-
db4/build_unix/.libs' ...
  findlibs: ... FOUND 'libdb-4.8.a' in '/home/umka/Work/colibri/colibri-
db4/build_unix/.libs'
```

8.3.6. Stack traces

Stack traces are useful to clearly see where build tool stumbled over an issue. In scons this may be done the way like this:

```
scons --debug=stacktrace
```

8.3.7. Making decisions

It is also useful to check how does build system makes decisions on building, searching , etc. Here is how it is done in scons:

```
scons --taskmastertrace=trace
```

The value passed to --taskmastertrace option is name of a file to store the trace. Trace shows detailed information on why build system changed dirs, built a targets, etc.

8.3.8. Configure issues

The most common set of issues to consider is configure issues. The package cannot find headers or libraries, etc. Complete information on how particular header was checked, what commands were executed is located in config.log file, which is usually placed in the project root directory.

Suppose the following scons execution error:

```
[umka@victim colibri-core]$ scons
scons: Reading SConscript files ...
Building at: build/x86_64.victim.localdomain
Checking for C header file db_int.h... no
db_int.h must be installed!
```

When we see something like this, we first of all look at config.log file. In this particular case it looks like this:

```
[umka@victim colibri-core]$ cat ./config.log
file /home/umka/Work/colibri/colibri-core/SConstruct,line 21:
Configure(confdir = .sconf_temp)
scons: Configure: Checking for C header file db_int.h...
.sconf_temp/conftest_0.c <-
  |
  |#include "db_int.h"
  |
  |
gcc -o .sconf_temp/conftest_0.o -c -O2 -Iinclude -I. .sconf_temp/conftest_0.c
.sconf_temp/conftest_0.c:2:20: error: db_int.h: No such file or directory
scons: Configure: no
```

As you can see from the log, headers path does not contain db4 paths and this is why db_int.h files was not found.

9. References

[References to all external documents (specifications, architecture and requirements documents, *etc.*) are placed here. The rest of the document cites references from this section. Use Google Docs bookmarks to link to the references from the main text.]

[0] Scons
[1] Autoconf
[2] Automake
[3] CMake

10. Inspection process data

[The tables below are filled in by design inspectors and reviewers. Measurements and defects are transferred to the appropriate project data-bases as necessary.]

10.1. Logt

Inspector	Task	Phase	Part	Date	Planned time (min.)	Actual time (min.)	Comments
Shadow	build_env	HLDINSP	1	3/2/2010	-	120	-
Nikita	build_env	HLDINSP	1	3/5/2010	-	80	-

10.2. Logd

No.	Task	Summary	Reported by	Date reported	Comments
1	build_env	Missed kernel code build in requirements and overall HLD	umka	2/23/2010	Fixed in HLDR
2	build_env	Mixed up functional and logical specs	umka	2/22/2010	Fixed in HLDR
3	build_env	Scons performance is not ideal as it comes to be. Granted it "high" for performance in comparison table, which is definitely wrong.	umka	2/22/2010	Fixed in HLDR
4	build_env	Missed to mention that scons supports platform-independent file system manipulation API. This makes it portability even better.	umka	2/22/2010	Fixed in HLDR
5	build_env	Missed one concurrency use case in "Concurrency control" section.	umka	2/22/2010	Fixed in HLDR

6	build_env	Did not mention distributed builds	umka	3/3/2010	Fixed in HLDINSP
7	build_env	Missed to mention CMake in alternatives	shadow	3/3/2010	Fixed in HLDINSP
8	build_env	Missed to mention about distributing scons with package	umka	3/3/2010	Fixed in HLDINSP
9	build_env	Missed to discuss linking against not installed libraries in "Refinement" section	umka	3/3/2010	Fixed in HLDINSP
10	build_env	Terminology ("parallel build") and typos	nikita	3/5/2010	-
11	build_env	Flexible module configuration options are needed	nikita	3/5/2010	-
12	build_env	Links to repository from commit messages	nikita	3/5/2010	-
13	build_env	Replace feature comparison table with use case based analysis	nikita	3/5/2010	-
14	build_env	File system requirements (case sensitivity, hard-links, symlinks)	nikita	3/5/2010	-
15	build_env	Justification for multiple repositories	nikita	3/5/2010	-
16	build_env	Db4 standard use case	nikita	3/5/2010	-
17	build_env	Security model	nikita	3/5/2010	-
18	build_env	User training is a part of deployment	nikita	3/5/2010	-