CIPRES build notes

Mark Holder, Terri Schwartz, and Rutger Vos

September 6, 2006

Contents

1 Quick Start

1.1 Quick start building on Unix-like platforms

1.1.1 Automatically build all dependencies

CIPRES-plus-deps.tar.gz, an archive of all of the non-standard libraries need by CIPRES can be downloaded from http://www.phylo.org/software/maintainer (if you want the snapshot of the code instead using svn to get the code, then use the http://www.phylo.org/software/SDK page).

After downloading the archive you should be able to build by simply performing the following steps:

```
tar xfvz CIPRES-plus-deps.tar.gz
cd CIPRES-and-deps
python configureCIPRES-and-deps.py
```

The subdirectory that is labelled cipres (or cipres with a version number if you downloaded a snapshot) will be your $CIPRES_TOP, and the build directory within it will be your CIPRES build directory.

1.1.2 Manual builds

These instructions are written assuming that $CIPRES_TOP is the top directory of the CIPRES source code distribution. To mimic the binary installation (which will be supported on Mac and Windows), we install omniORB and omniORBpy inside $CIPRES_TOP/build

1. From the top of the omniORB-4.0.7 distrubitution:

```
./configure --prefix=$CIPRES_TOP/build --disable-static
make
make install
```

2. From the top of the omniORBpy-2.7 distrubitution:

```
./configure --prefix=$CIPRES_TOP/build --with-omniorb=$CIPRES_TOP/build
make
make install
```

Once the dependencies are built, you are ready to configure and build cipres:

1. In `$CIPRES_TOP` run `boostrap.sh` (you have to do this whenever `$CIPRES_TOP/configure.ac` or the macros in the `$CIPRES_TOP/config/m4` change).
2. Verify that you have the `boost` library
3. Run the configure script. Something like:

```
./configure --prefix=`pwd`/build --with-omniorb-prefix=<path to omniORB> --
   with-boost=<path to boost> --enable-end-user-dist
```

   Use the `--with-boost` if you have installed the `boost` library. If you don't want install `boost`, you can just download the libraries and set `$BOOST_ROOT` to the path to the directory that named `boost_1_33_x`. If you are relying on `$BOOST_ROOT`, then do not use the `--with-boost=...` arg to `configure`. Use `configure --help` to see all of the optional arguments and how to tell `configure` where your dependencies are.

4. `make`
5. `make install`

1.2 Quick start building on Windows

1.2.1 Automatically build all dependencies

`cipres-and-deps-win32.zip`, an archive of all of the non-standard libraries need by CIPRES can be downloaded from http://www.phylo.org/software/maintainer.

After downloading the archive you should be able to build by simply performing the following steps:

```
jar xvf cipres-and-deps-win32.zip
cd cipres-and-deps
```

See `cipres-and-deps/readme.txt` and set environment variables as it instructs.

```
configureTools.bat
cipres\cipres-build-scripts\build-for-windows\makeCipres.bat
cipres\cipres-build-scripts\build-for-windows\installCipres.bat
```

The subdirectory that is labelled `cipres` will be your `$CIPRES_TOP`, and the `build` directory within it will be your CIPRES build directory.

1.2.2 Manual builds

This section needs to be fleshed out. Basically you have to create a custom `buildConfig.bat` file in the parent of `$CIPRES_TOP`. You can use `$CIPRES_TOP/cipres-build-scripts/build-for-windows/exampleBuildConfig.bat` as a template. After this file is configured, you can go to the `$CIPRES_TOP/cipres-build-scripts/build-for-windows` directory and run:

```
makeCipres.bat
installCipres.bat
```

after any code change.

2 Introduction

This document explains the process of building the CIPRES software framework and core modules from source. In August 2006, CIPRES moved from a home-made, python-based build system to an `autoconf`-based procedure.

In this document, we will distinguish between:

- **maintainers** – developers who are members of the CIPRES-dev team. These folks are expected to keep the build system working.
- **builders** – people who want to download and build[1]
- **programmers** – builders who may want to write code that uses the CIPRES library, but who will not be expecting to commit to the CIPRES SVN repository, change the IDL, or edit build configuration files. Programmers will have the same needs as builders, but will also need a consistent API for using the CIPRES library.
- **users** – folks who are not expected to compile the software (Mac and Windows only). Administrative/superuser privileges are only expected for multi-machine installations (and even then are only required if the installation is going to a write-protected location, or port-forwarding is needed to bypass firewalls).

2.1 Installation constraints

CIPRES is difficult primarily because the end product is a diverse set of modules instead of a single application. The CIPRES framework is a complex and many components require information that will depend on how the system was installed and the current preferences of the user (see Table 1). The <> notation is used to indicate paths that are determined by CIPRES components – usually by checking multiple locations in a predefined order (see Table 2). Note that it is most critical that the CIPRES Registry be able to find runtime paths without the user of environment variables. The CIPRES Registry can provide environmental variables to the processes that it launches. Thus it is safe for a service to require an environmental setting if

1. the service is always launched by the CIPRES Registry and the CIPRES Registry can determine the correct setting; or

[1] We are only planning on binary installations of CIPRES for Mac and Windows, so all Linux users will be builders as well as users

2. it is reasonable to require the users of the service to set the variable (e.g. writers of some command-line tools might feel confident that their users are sophisticated enough to know how to set an env. variable)

We are trying to make the CIPRES Registry as flexible as possible. For example when new services are installed they specify how they should be launched and can define new properties. These properties can be set in collaboration with the user and can hold installation-specific information such as the path to a helper executable. Despite this, we will probably need to be pretty rigid with respect to the relative location of the core components. This rigidity conflicts with the autoconf/automake desire to let users and sys admins determine where different components end up. It is not clear at this point, how easy it will be for CIPRES to tolerate having its components strewn all over a user's hard drive.

Table 1: Example of runtime information needed: CIPRES Registry application

Information needed	Solution
Path to the CIPRES jar files	The relative path from CIPRES Registry launching assumed to be constant. The script for launching the CIPRES Registry determines its location, and then supplies the relative path to the jar file directory.
Which CIPRES services are installed	reads CIPRES Registry XML files and properties in the `<CIPRES_INSTALL_DIR>/share/cipres/resources` services directory.
User's preferences	Must find and read properties file. Checks for `cipres_config.properties` in `<CIPRES_USER_DIR>`. If neither is found, then one is created by copying from the `cipres_config.properties` file from `<CIPRES_INSTALL_DIR>/share/cipres/resources` .
Location of installed dependencies (e.g. PAUP*)	Stored in user preference file.

Table 2: Shorthand for paths (the use of `$` indicates an environmental variable)

Notation	Purpose	Cascade of possible locations
`$CIPRES_TOP`	Top of your local copy of CIPRES SVN archive	`$CIPRES_TOP`
`<CIPRES_USER_DIR>`	Home of user-controlled properties (e.g. logging level)	1. `$CIPRES_USER_DIR` 2. `$HOME/cipres`
`<CIPRES_INSTALL_DIR>`	Top of installed CIPRES components (shared by all users of a machine)	1. `$CIPRES_ROOT` 2. Java code looks on the classpath for `cipres_config.properties`. If the file is found, it is assumed to be in `<CIPRES_INSTALL_DIR>/share/cipres/resources` .

2.2 The goal

I'm hoping that we can maintain the same directory structure on all platforms. On Mac and Windows we should be able to keep the gory details hidden in a directory that the user does not need to look into. A double-clickable application bundle would know how to find this directory and fire up CIPRES.

The directory structure that I think we should shoot for is a compromise between our previous "`cipres_dist`" directory (shown in Table 3).

Table 3: Directory structure of built CIPRES

Property	Default Path	Contents
`cipres.bin.path`	`$CIPRES_ROOT`/bin	core executables (e.g. `cipres_java.sh`, `read_nexus_server`)
`cipres.shared.lib.path`	`$CIPRES_ROOT`/lib/cipres	shared object code that is dynamically linked into CIPRES applications (e.g. `libcipres`
`cipres.extern.lib.path`	`$CIPRES_ROOT`/lib	shared object code that is dynamically linked into CIPRES applications (e.g. `libomniORB4`
`cipres.python.lib.path`	`$CIPRES_ROOT`/lib/python/site-packages	location of CIPRES python modules
`cipres.perl.lib.path`	`$CIPRES_ROOT`/lib/perl	location of CIPRES perl modules
`cipres.data.path`	`$CIPRES_ROOT`/share/cipres	location of miscellaneous files from the installation (`help`, `xml`, etc.)
`cipres.resources.path`	`cipres.data.path`/resources	master version of the properties file and configuration files
CIPRES system-wide services	`cipres.resources.path`/services	information about installed packages.

3 Required Build Tools (Maintainers)

3.1 GNU Autotools

System for writing "portable" Makefiles and code (note "portable" here does not refer to Windows unless you are using Cygwin). `autoconf` and `automake` help you write a `configure` script and `Makefile.in` files. When a user downloads the source and runs the `configure` script, `Makefile`s with appropriate settings for the OS, architecture, and user settings are produced. This configuration process also produces a C/C++ header file called `config.h` that is included during C/C++ compilation. This file contains C preprocessor directives and definitions for features that are needed in the code, but not standardized by the C or C++ languages.

Tutorial: http://autotoolset.sourceforge.net/tutorial.html
Resources: http://www.gnu.org/software/autoconf/manual/autoconf-2.57/html_node/autoconf_194.html

- `autoconf` version 2.5.9

- `automake` version 1.9.6
- `autoreconf` version 2.5.9

`autoreconf` calls `autoconf` and the other tools in the right order. Thus, maintainers should run `autoreconf` whenever files involved in creating the `configure` script change.

3.2 Known Problems

NOTE: `aclocal` is an alias to `automake`. The version must be greater than 1.6.3 (1.9.6 works intermediate versions are not checked) for `autoreconf` to succeed. If you do not have the correct version, no warning will be issued (the error seems occurr before `autoconf`'s version checking macros are enforced. The error text refers to missing Python macros that are supplied in 1.9.? versions of `automake` The errors will look like:

```
aclocal: macro ‘PYTHON_SITE_PKG’ required but not defined
aclocal: macro ‘PYTHON_VERSION’ required but not defined
aclocal: macro ‘PYTHON_VERSION’ required but not defined
aclocal: macro ‘PYTHON_SITE_PKG’ required but not defined
aclocal: macro ‘PYTHON_VERSION’ required but not defined
```

4 Required Build Tools (Programmers)

- `ant`
- `g++`
- `make`
- `perl`
- `python`

5 Third-party libraries

5.1 `omniORB` (C++ ORB)

Build and install `omniORB` (4.0.6 or later)

```
configure --prefix=<your preferred install dir> --disable-static
make
make install
```

Add `omniidl` to your `$PATH`

5.2 omniORBpy (Python ORB)

Build and install omniORBpy (version that corresponds to you omniORB)

```
configure --prefix=<your preferred install dir> --with-omniorb=<prefix dir for
    omniORB installation>
make
make install
```

```
'configure␣--prefix=<your␣preferred␣install␣dir>␣--with-omniorb=<prefix␣dir␣
   for␣omniORB␣installation>'
# comment
make
make install
```

5.3 boost (C++ library)

Build and install boost (1.33.0 and 1.33.1 have been tested).

```
bjam --prefix=<your preferred install dir> ``sTOOLS=<your compiler's code>''
   install
make
make install
```

Note that the compiler codes in the boost documentatin (e.g. when using gcc on Mac the $TOOLS should be darwin, *not* gcc.

6 Building CIPRES

```
config/autogen.sh
./configure --prefix=<your preferred install dir> --with-omniorb-prefix=<
   prefix passed to omniORB configure> --enable-documentation --with-boost=<
   prefix passed to bjam when building boost>
make
make install
```

6.1 Using eclipse

Parts of CIPRES are written in Java, C++, Python and Perl. On Mac and *nix systems, we use an autoconf build system that delegates parts of the build to language specific tools (ant for Java, setup.py for Python, etc). On Windows, the build process is coordinated by batch files.

Even if you're only going to be working on the Java parts of CIPRES, you need to build the full system and install it as explained above (or see http://www.phylo.org/software/maintainer). After successfully building, you can start using eclipse.

To use eclipse import the CIPRES-Library project:

1. `File→Import→Existing Projects Into Workspace`
2. click "`Next`"
3. Select "`Root Directory`"
4. Browse to the `$CIPRES_TOP` directory
5. Choose the CIPRES-Library project

6.1.1 Caveats

You're still going to need to use `ant` periodically for two reasons:

1. If any of the .idl files change, source code needs to be re-generated and this isn't handled within the eclipse project
2. The recidcm3 service is written in Java and is launched in a separate JVM by the registry. The registry launches it from `$CIPRES_TOP/build/lib/cipres/cipres-1.0.1.jar` which is built from the code in the cipres-library-jar directory. When you use eclipse the jar files aren't updated. To make sure the jars are up to date
   - on *nix: simply run `make` and `make install` from a `$CIPRES_TOP`.
   - on Windows: go to the parent of `$CIPRES_TOP` and run `buildConfig.bat` to set environment variables. Then:

```
cd cipres/framework/java/cipres-idl-jar
ant jar install
cd cipres/framework/java/cipres-library-jar
ant jar install
```

7 Problems

8 Supported platforms

Platforms on which we have built/tested:

Hostname	arch	OS	C++ compiler	Java	Python	Builds	Runs
petal	x86_64	linux2	g++ 4.0.1	1.5.0_05	2.3.4	yes	?
petal040	x86_64	linux2	icc 8.0	1.5.0_05	2.3.4	no (python distrib. broken)	?
cheefour	ppc	darwin	g++ 3.3	1.4.2_09	2.4	yes	?
grove	ppc	darwin	g++ 3.3	1.4.2_09	2.4.1	no (link errors for dcm)	?
tempest	sun4	sunos5	g++ 3.3.2	1.4.2_08	2.3.3	yes	?

9 Documentation

This document was written in LaTeX. Source for the entire document is `$CIPRES_TOP/doc/notes.tex` Content for each section (or subsection) comes from a separate file (in `$CIPRES_TOP/doc/latex`). This separation was done to make it

easy to include and exclude sections in multiple different documents (e.g. we may want Java only documentation for developers uninterested in Python or C/C++), and also to reduce the TEXcompilation times.

9.1 Why LATEX

Primarily because MTH does not know how to use Microsoft Word™, and he is being a jerk about not wanting to learn it. Advantages of LATEX:

- Macros in `$CIPRES_TOP/doc/latex/preamble/.tex` helps the source code become richer and allows for some separation between content and display.
- Plain text format (with return characters between every sentence) make source code management easier (fewer manual merges).
- Resulting document looks nice

9.2 The listings package

The `listings` package (available from http://mirror.aarnet.edu.au/pub/CTAN/macros/latex/contrib/listings/) is being used to format snippets of code. MTH has wrapped some of the functionality of listings command in envronments (e.g. `$Python`, `$shell`, `$Makefile`) in `$CIPRES_TOP/doc/latex/preamble.tex`. This system gives us not only syntax highlighting, but (more importantly) the ability to copy and paste from a script into the LATEXdocumentation without needing to escape characters like $.

10 Deprecated tools

10.1 automake-idl

Notes on `automake-idl`, a patch to automake to add support for configure-time ORB substitution. The package was brittle (failing to work with MTH's omniORB installation, for example), and it is unlikely that we will use the functionality (at minimum switching ORB's will require some testing of the current CIPRES library code – not just a command line switch to `configure`). The files in `$CIPRES_TOP/config/m4`) with names of the form `ai_(ORB-NAME).m4` are tweaked versions from the `automake-idl`. Only `ai_omniorb.m4` and `ai_omniorbpy.m4` have been tested.

Downloaded and installed `automake-idl` (from http://autotools-idl.sourceforge.net/). This is a patch of `automake` to deal with IDL compilation. This must be installed so that it is found when you invoke `automake-1.9`

11 Previously observed problems

These (seem) to be corrected, but documentation is retained in case they crop up again.

11.1 omniidl on $PATH

The first time I ran `configure` for CIPRES, it complained that `omniidl` was not on my path.
At later points (after removing `omniORB` from system locations and about a million other changes) the `configure` did not complain. However compilation of IDL to Python is failing for me (MTH) and in the `$CIPRES_TOP/framework/python/Makefile` I have:

```
IDLC = PATH=/Users/mholder/installed/omniORB-4.0.7/bin:$$PATH
   DYLD_LIBRARY_PATH=/Users/mholder/installed/omniORB-4.0.7/lib:
   $$DYLD_LIBRARY_PATH omniidl
...
OMNIIDL =
```

and `OMNIIDL` is used later in the Makefile. Terri has `OMNIIDL=<path to her \omniidl>` in her Makefile (and the python builds for her).

References