Distributed Builds with Rendezvous and DistCC

Vadim Zaliva, lord@crocodile.org

June 3, 2004

Contents

1 Distributed compilation

If you are working on a sizable C, C++ or Objective C project and compilation time becomes an issue, you may want to consider organizing a compile farm and running a distributed compile on several machines simultaneously. There is an excellent product named DistCC which allows you to do this.

The idea is that you install the distcc daemon on multiple machines which will become your "compile farm". When you compile from your client machine, it distributes the work (per source file) to multiple machines in your farm.

DistCC works as a wrapper around the compiler. It passes source files through the preprocessor (thus eliminating all local dependencies) and sends the resulting pre-processed source to one of the build farm machines over the

network using TCP protocol. The build machine compiles the received files and sends resulting object files back to the client. Since the files are already pre-processed, the build farm compiler does not need to have any header files or know any macro definitions which you might need to compile the original source. This way, you do not need to install libraries and header files of 2rd party libraries on each of the compile farm machines - you only need them on your client machine.

All linking is done on the client machine. To parallelize the build process, it is recommended that you use the GNU make "-j" option. When it is specified, GNU make tries to compile several files in parallel (if there are no dependencies among them). With DistCC they would end up being sent to different compile farm machines and compiled in parallel.

2 Automating build farm machines discovery using Rendezvous

The main problem with simple DistCC deployment is that you need to know the names of all the machines in the compile farm. Usually you need to list them in the DISTCC_HOSTS environment variable. If some hosts become unavailable, or new hosts are added, you need to know and must update your local configuration accordingly. In a larger organization, this quickly becomes a time consuming issue.

In their latest development tools release, Apple also makes use of a distributed build feature. They populate the list of machines in a compile farm automatically by using Rendezvous technology. Interestingly, they are using DistCC as well as actual build distribution technology.

In this article we will show how you can implement something similar in Linux, giving you all the benefits of a distributed build farm with automatic discovery while using inexpensive hardware and free software. We will describe the setup for this using Red Hat Linux 9, but with minimal changes it could be used with other Linux and Unix platforms as well.

3 Setting up compilation farm servers

First we need to Rendezvous-enable your Linux boxes. There are several implementations of Rendezvous for Linux, even one from Apple. Rendezvous is in fact Apple's name for Zeroconf technology. We have chosen an open source cross-platform implementation of Zeroconf for Linux called "HOWL" from Porchdog Software.

First, download and install HOWL from Porchdog Software's web site. We used the RPM package they provide.

HOWL distribution includes several daemons, but only one, namely "mDNSResponder" (dynamic publication and service discovery daemon) is of interest to us.

After it has been installed, you need to make sure it will start automatically on boot:

```
chkconfig --add mDNSResponder
chkconfig --level=345 mDNSResponder on
```

howl-0.9.5-1.i386.rpm has a small mistake in the file "/etc/init.d/mDNSResponder". It specifies the wrong path to "mDNSResponder binary". Also, it does not specify the configuration file's location. After installation, apply this small patch to "/etc/init.d/mDNSResponder" to correct these problems:

```
--- mDNSResponder.old   2004-06-02 17:16:03.000000000 -0700
+++ /etc/init.d/mDNSResponder   2004-06-02 17:45:47.000000000 -0700@@ -12,7 +12,7 @@
 # processname: mDNSResponder
 # config:

-OTHER_MDNSRD_OPTS=""
+OTHER_MDNSRD_OPTS="-f /etc/mDNSResponder.conf"

 # Source function library.
 . /etc/init.d/functions
@@ -24,7 +24,7 @@

 start() {
        echo -n $"Starting mDNSResponder... "
-       /usr/local/bin/mDNSResponder $OTHER_MDNSRD_OPTS
+       /usr/bin/mDNSResponder $OTHER_MDNSRD_OPTS
        RETVAL=$?
        echo        return $RETVAL
```

Now you need to create a configuration file with a list of the services on this machine, which you want to advertise via Rendezvous. For example "/etc/mDNSResponder.conf/" may look like this:

```
# mDNSResponder conf file
#
#       name  type              domain  port    text record
        rover _ssh._tcp        local.   22
        rover _distcc-RH9._tcp local.   3632
```

This will advertise SSH and DISTCC services on this machine. In the example above, 'rover' is a local machine name. Replace it with the actual name of your host. Since all machines in a DistCC compile farm must have the same architecture and name for the compiler, we called our service "_distcc–RH9" to show that this is Red Hat 9 i386 machine. You may run several farms in the same organization by using different names, for example "_distcc–Solaris8" or "_distcc–FreeBSD5".

Now you can start "mDNSResponder" with the following command:

```
service mDNSResponder start
```

If you have a MacOS X machine on the local network you can test how this works by using a Terminal application. Go to "File–Connect To Server" menu and you should see the name of your Linux machine in the list of SSH hosts there. For example:

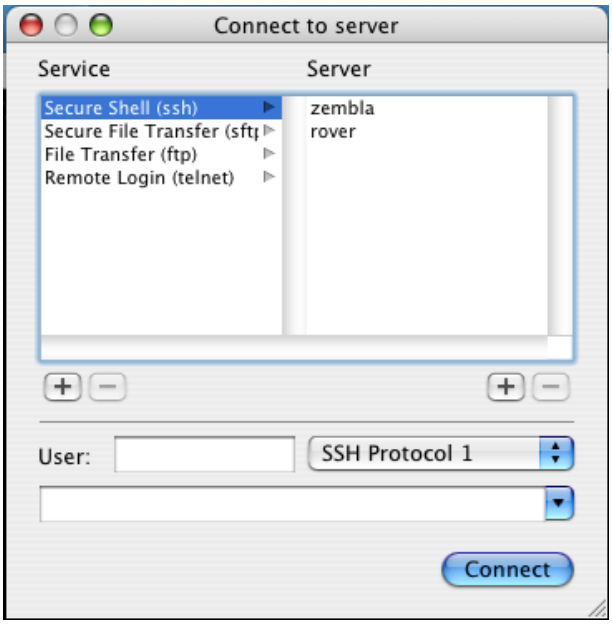

Figure 1: "Connect To Server" dialogue of MacOS X Terminal program

Now you need to install distcc-server. (We used RPMs for Red Hat Linux 9)

Make sure it will start on boot:

```
chkconfig --add distcc
chkconfig --level=345 distcc on
```

And start it:

```
service distcc start
```

Repeat these steps for all compilation farm machines.

4 Setting up compilation farm clients

To use Rendezvous on clients, you will also need to install and run the "mDNSResponder daemon". Follow the steps from the previous section to install it. However this time you should not include the configuration line for distcc in "mDNSResponder.conf" unless your client machine also acts as a compilation server.

Next we need to compile a small program which we will use to look up distcc servers using Rendezvous. To compile it you need to install HOWL development libraries and headers on one of the machines. (You can do this on one machine and copy the resulting binary to all others). We used howl-devel RPM package to install HOWL libraries.

The source code of "mDNSlookup.c" could be found in Appendix B on page 8. Compile it with the following command:

```
cc -I/usr/include/howl mDNSlookup.c -o mDNSlookup -lhowl -lpthread -lrt
```

And install in your executable path (for example to "/usr/local/bin/").

Now, finally we can test to see what machines we can discover:

```
[root@rover root]# mDNSlookup _ssh._tcp 500
rover _distcc-RH9._tcp. local. 192.0.2.140 3632
zembla _distcc-RH9._tcp. local. 192.0.2.139 3632
```

In the example above, it found 2 machines: "zembla" and "rover" with IP addresses 192.0.2.139 and 192.0.2.140, respectively. Both machines are offering compilation services using DistCC on Red Hat 9, listening on TCP port 3632.

5 Setting up your compiler

Now we just have one last step left: to make use of discovered hosts. There are several ways to do this. We have chosen the compiler shell wrapper approach. The main advantages of this approach are:

- On multi-user systems users can control individually when DistCC should be used.
- There is no need to modify project Makefiles to take advantage of distributed builds.
- Users can easily switch distributed builds off and on (some tools, for example "imake", do not work well when DistCC use is enabled).

You need to install some scripts listed in Appendix A on page 6 in some directory which is in the binary executable path of all users (e.g. "/usr/local/bin/"):

You may want to tweak the last two scripts a bit ("distcc_nhosts" and "distcc_wrapper"), adjusting the variables at the beginning. The variable MDNS_TIMEOUT defines how long to wait for Rendezvous responses (in milliseconds). ADD_HOSTS_BEFORE and ADD_HOSTS_AFTER allow you to add some non-Rendezvous-enabled compile farm hosts to list hosts which will be used. The value of SERVICE_NAME should match the one you set earlier in "/etc/mDNSResponder.conf" on the compile farm hosts. To verify that it works you may temporary uncomment the line, setting the DISTCC_VERBOSE variable. This would produce compilation-time debug output from distcc showing what hosts are used.

There is another product which could be used together with distcc to speed up compilation even further: CCACHE. It works pretty well together with DistCC and provides a noticeable compilation speed up, even when using a compilation farm. If you have it installed (you need to install it just on the client machine), replace the following line in "distcc_wrapper":

```
exec distcc /usr/bin/c++ $*
```

with

```
exec ccache distcc /usr/bin/c++ $*
```

Now you are all set. To start your first distributed compile, just type:

```
make -j ‘distcc_nhosts‘
```

6 Conclusions and Further Work

We have shown how to build an inexpensive and robust solution for distribute Linux builds across multiple machines. Once set up, no changes need to be made on the client side when new machines are added to or removed from a farm. If some machine becomes temporary unavailable (for instance due to network connectivity issues), this will be immediately discovered and clients will make no attempt to use it.

If a number of machines in the compile farm as well as a number of client machines are noticeable, it might make sense to prepare RPM packages for both the client and sever which will eliminate any manual configuration, except RPM installation. "mDNSResponder.conf" could be generated automatically by the RPM post-install script (with the name of the machine taken from the "hostname" command and the service name build from the machine architecture name and gcc version).

A minor limitation of the current approach is that DistCC service is advertised whenever a machine is up, regardless of whether the DistCC daemon is running or not. This could be corrected fairly simply by a small modification of distcc daemon to register on "mDNSResponder" on startup and unregister on exit. HOWL provides an example of such code in the source code of "mDSNPublish" example.

A Client Scripts Source Code

A.1 distcc_on

```
#!/bin/sh
# Enables distributed compilation using DistCC
ln -s /usr/local/bin/distcc_wrapper ~/bin/c++
ln -s /usr/local/bin/distcc_wrapper ~/bin/cc
ln -s /usr/local/bin/distcc_wrapper ~/bin/g++
ln -s /usr/local/bin/distcc_wrapper ~/bin/gcc
```

A.2 distcc_off

```
#!/bin/sh
# Disables distributed compilation using DistCC
rm -f ~/bin/c++ ~/bin/cc ~/bin/g++ ~/bin/gcc
```

A.3 distcc_status

```
#!/bin/sh
# Shows if distributed compilation using DistCC is enabled
if [ -f ~/bin/c++ ] ; then
    echo "distcc is ON"
else
    echo "discc is OFF"
fi
```

A.4 distcc_nhosts

```
#!/bin/sh
# Prints number of hosts availiabe in compile farm
# Output of this script intended to be used with -j option
# of GNU make.
#
# By default we return number twice as much as hosts found in compile
# farm, plus one (to handle empty compile farm).

MDNS_TIMEOUT=200
SERVICE_NAME=_distcc-RH9._tcp

ADD_NHOSTS=1
MULTIPLY_NHOSTS=2

echo $(($ADD_NHOSTS+$((`mDNSlookup "$SERVICE_NAME" $MDNS_TIMEOUT  | wc -l`+$MULTIPLY_NHOSTS))))
```

A.5 distcc_wrapper

```
#!/bin/sh
#
# DistCC wrapper to use Rendezvous to discover compile
# farm machines. Should not be run directly by user.

MDNS_TIMEOUT=200
ADD_HOSTS_BEFORE=
ADD_HOSTS_AFTER=
SERVICE_NAME=_distcc-RH9._tcp

DYNAMIC_HOSTS=`mDNSlookup "$SERVICE_NAME" $MDNS_TIMEOUT | awk '{printf "%s:%s ",$4,$5;}'`
export DISTCC_HOSTS="$ADD_HOSTS_BEFORE $DYNAMIC_HOSTS $ADD_HOSTS_AFTER"
export DISTCC_NHOSTS=`echo "$DISTCC_HOSTS" | wc -w`
echo $DISTCC_HOSTS
```

```
#export DISTCC_VERBOSE=1
exec distcc /usr/bin/c++ $*
```

B mDNSlookup Source Code

```
/*
 * Copyright 2003, 2004, 2004 Porchdog Software. All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *      1. Redistributions of source code must retain the above
 *      copyright notice, this list of conditions and the following
 *      disclaimer.
 *
 *      2. Redistributions in binary form must reproduce the above
 *      copyright notice, this list of conditions and the following
 *      disclaimer in the documentation and/or other materials
 *      provided with the distribution.
 *
 *  THIS SOFTWARE IS PROVIDED BY PORCHDOG SOFTWARE ``AS IS'' AND ANY
 *  EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 *  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 *  PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE HOWL PROJECT OR
 *  CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 *  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 *  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
 *  USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
 *  AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *
 *  The views and conclusions contained in the software and
 *  documentation are those of the authors and should not be
 *  interpreted as representing official policies, either expressed or
 *  implied, of Porchdog Software.
 *
 * ----------------------------------------------------------------------
 *
 * mDNSlookup is quick hack, based on mDNSBrowse example from HOWL package.
 *
 * Main changes:
 * 1. introduced timeout parameter
 * 2. less-verbose output (do not report some packets, do not print TXT records)
 *
 * To compile on Linux use following command:
 *
 * cc -I/usr/include/howl mDNSlookup.c -o mDNSlookup -lhowl -lpthread -lrt
 *
 * Vadim Zaliva
 *
 */

#include <howl.h>

#include <stdio.h>
#include <time.h>

static sw_result HOWL_API lookup_resolver(
    sw_discovery_resolve_handler    handler,
    sw_discovery                    discovery,
    sw_discovery_resolve_id         id,
    sw_const_string                 name,
```

```
    sw_const_string                 type,
    sw_const_string                 domain,
    sw_ipv4_address                 address,
    sw_port                         port,
    sw_const_string                 text_record_string,
    sw_octets                       text_record,
    sw_ulong                        text_record_len,
    sw_opaque                       extra)
{
    char name_buf[16];

    sw_discovery_stop_resolve(discovery, id);
    printf("%s %s %s %s %d\n", name, type, domain,
           sw_ipv4_address_name(address, name_buf, 16), port
    );
    return SW_OKAY;
}


static sw_result HOWL_API lookup_browser(
    sw_discovery_browse_handler     handler,
    sw_discovery                    discovery,
    sw_discovery_browse_id          id,
    sw_discovery_browse_status      status,
    sw_const_string                 name,
    sw_const_string                 type,
    sw_const_string                 domain,
    sw_opaque                       extra)
{
    sw_discovery_resolve_id rid;

    if(status==SW_DISCOVERY_BROWSE_ADD_SERVICE)
    {
        if(sw_discovery_resolve(discovery, name,
                                type, domain,
                                NULL, lookup_resolver,
                                NULL, &rid) != SW_OKAY)
        {
            fprintf(stderr, "resolve failed\n");
            return -1;
        }
    }
    return SW_OKAY;
}


main(int argc, char **argv)
{
    sw_discovery            discovery ;
    sw_result               result    ;
    sw_discovery_browse_id  id        ;
    sw_salt                 salt      ;
    sw_ulong                timeout   ;

    if(argc == 3)
    {
        char *e;
        timeout=strtol(argv[2], &e, 10);
        if(*e)
        {
            fprintf(stderr, "Invalid timeout value. Should be number in milliseconds!\n");
            fprintf(stderr, "Usage: mDNSlookup <type> [timeout]\n");
            return -1;
        }
    } else if(argc != 2)
    {
        fprintf(stderr, "Usage: mDNSlookup <type> [timeout]\n");
        return -1;
```

```
    } else
        timeout = 0L;

    if((result = sw_discovery_init(&discovery)) != SW_OKAY)
    {
        fprintf(stderr, "sw_discovery_init failed: %d\n", result);
        return -1;
    }

    if(sw_discovery_browse(discovery, argv[1], NULL,
                           NULL, lookup_browser, NULL, &id) != SW_OKAY)
    {
        fprintf(stderr, "sw_discovery_browse_services failed: %d\n", result);
        return -1;
    }

    if(sw_discovery_salt(discovery, &salt)!= SW_OKAY)
    {
        fprintf(stderr, "sw_discovery_salt failed: %d\n", result);
        return -1;
    }

    if(timeout == 0L)
    {
        /* No timeout - wait forever. */
        sw_discovery_run(discovery);
    } else
    {
        /*
          Correct code should be:

          while(timeout != 0L)
          sw_salt_step(salt, &timeout);

          but sw_salt_step() does not modify
          timeout value (this bug was reported
          to authors).
        */

        /*
          So, we have to do a workaround, using
          POSIX 1003.1b Section 14 (Clocks and Timers) API:
        */

        struct timespec start;
        clock_gettime(CLOCK_REALTIME, &start);
        while(1)
        {
            struct timespec now;
            clock_gettime(CLOCK_REALTIME, &now);

            long waited = (1000L*now.tv_sec+now.tv_nsec/1000000L) -
            (1000L*start.tv_sec+start.tv_nsec/1000000L);

            if(waited>=timeout)
                break;

            sw_ulong remain=timeout-waited;
            sw_salt_step(salt, &remain);
        }
    }
    return 0;
}
```