Dansarina

Daniel C. Bastos
dbastos@math.utoledo.edu

Nov 1, 2005

Abstract

This document presents a few details of dansarina. Dansarina is an IRC robot written for portability. We describe in this document a few pieces of the source code in details, but, mainly, we provide general information about the development, including what has not yet been finalized, and all the problems we have been facing.

1 Introduction

Dansarina is an IRC robot. She's composed of a small collection of programs which help each other to accomplish all the tasks that she performs. She has been written to be portable and simple. She has a very simple user interface and very easy to use — see the installation and user interface sections for how to get her running for the first time — not written yet; there is no installation yet, and no user interface anyway.

Dansarina has a plugin system which allows programmers — that can write `sh` and `awk` — to extend her in a variety of ways. The scripts can take advantage of the network with the help of auxiliary programs to handle the network connections — see the plugin system section for more information. So, dansarina is able to retrieve a paragraph from Wikipedia so that users can consult an encyclopedia while talking about a subject, she is able to consult an online dictionary, or perhaps perform a search in a database to answer how some Win32 API datatype is declared, et cetera. I have personally written these three plugins. But, the plugin system goes beyond that: with AWK, it's very nice to handle simple databases, and so dansarina is able to easily keep track of who's been in a channel, who has left, at what time, what did they say last, what was their quit message, et cetera. All of this and more can be changed or implemented by writing her plugins.

Dansarina can connect to other dansarinas to form a network of dansarinas. Through the network, users can exchange messages, talk, manage each other's dansarinas, et cetera.

The network is created through some particular plugins, but separately from the plugin system introduced above — this network is not yet created; the current network actually links only two dansarinas; not more than that. We want to distinguish between network plugins and regular plugins because the network requires a new kind of communication and bookeeping of information, so dansarina gets help with a different daemon — different than

the plugin daemon — which handles all the network tasks by calling appropriate network plugins.

1.1 History

I have kept a history of dansarina's development. We're currently in the 0.61 version, but you can still see most of the packages on my homepage, including the early versions 0.1, 0.2, et cetera.

If you look at each package individually you may see radical changes from one version to another, but such radical changes are only present in the early versions. For example, the first versions were written entirely in a "literate programming" model, but I gave up on that soon enough. The package has been without a nice documentation for most of its life, but on the version 0.60, I decided to bring back some of the "literate programming" philosophy. I say "some" because I do it in my own way now; my own way is called the "bit methodology" and is described in the "bit" package, on my homepage.

1.2 Overview of the source code

Dansarina is an application that can't really take advantage of the UNIX standard io library; on the contrary, it would be harder to use it — see, for example, the problems described "UNIX Network Programming" volume one, by Richard Stevens. So, instead, we use a small io library designed by Daniel J. Bernstein. I wrote small libraries for most of the other tasks. For example, to handle socket operations, I wrote the socket library; to handle the hassles of controlling network buffers I wrote the peer library. For databases, I wrote the db library. For strings, I wrote the str library — which is documented on my homepage; see my str package.

I believe that the most important information about dansarina is how the source code is arranged, and the path of the flow of the program. Dansarina's starting point is written in `dansarina.c`. Upon startup, she reads her configuration and initializes some data structures — for example, the peer array called `user`.

Every connection is considered a user, so even the listening sockets are attached to the `user` array, using the peer library. Dansarina distinguishes the IRC server from regular peers, so she process them separately — the IRC server has somewhat a higher priority.

Dansarina is somewhat event-driven. For example, when data comes in from the IRC server, dansarina analyzes the data and generates an event. She recognizes the events by one of her "dispatch" functions in `events.c`. In the same file, we have the `event` arrays which are lookup tables that inform dansarina who is responsible for which event — the lookup job is done by the function `find`. This is it.

1.3 Authentication of peers

Authentication is required when a DCC CHAT from a user is issued to dansarina, or when another dansarina wants to link herself to dansarina.

The peer library uses an array of `struct peer`. A peer structure groups the input and output io buffers, and it includes some variables for control. For example, if `peer.cont` is non-zero, then a line that was being read from the network has not completely arrived yet; so on a next read, data will be concatenated.

A peer goes through many stages upon connection. For example, when a "DCC chat" is issued, we already know the nickname of the user, so we set the bit `PEER_AUTH_USR` on the user's stage `unsigned long` variable. By invoking `peer_auth`, we'll read and check the user's password, then.

The stage level is interesting for linking robots, because robots go through many stages. A robot is connected only after all stages of the protocol have been successfuly terminated.

1.4 Databases

For detailed information on the database library, see the db package on my homepage; it describes the database interface and its limitations. The database has some serious limitations on performance, but no performance issue should affect very small databases.

Dansarina needs a user database; the `usr/` directory. There, you will find a record per user. The format of each record is easily parsed. The protocol is netstrings; see

http://cr.yp.to/proto/netstrings.txt.

User records are structured as follows: nickname, hostname, password, user flags, UNIX timestamp with creation date, UNIX timestamp with last seen date. Each user record is kept in memory in a `struct urecord` — see `user.h`.

2 The plugin system

The plugin system is composed of a program called `plugger`. Dansarina opens a pipe to `plugger` upon startup. If she can't start the daemon up and connect a pipe to it, she should log that information, and boot up normally. Of course, the plugin system would be disabled and users would find that out when they would ask for plugins. Currently, this is not implemented; right now, most errors are fatal; if Dansarina can't connect a pipe to `plugger`, she will just report that and end execution; if a DCC CHAT connection fails, she will also end execution. Handling of errors and user interface — such as reporting problems nicely, and logging — have not been done yet.

Plugins are almost any UNIX program, but Dansarina is limited to `sh` and `awk` programs, by convention. These programs, however, may use the program `tcpclient`, written by Daniel J. Bernstein in his UCSPI package, to make network connections — also notice that the GNU AWK has network support, but its use is probably discouraged.

The program `plugger` is a plugin daemon; it runs programs in the background and gives them a deadline of execution. If they don't meet the deadline, they are killed and users are — should be — notified. If they do, they will probably say something to their standard output which in turn makes `plugger` pass these informations back to Dansarina and she

would pass it back to the IRC server. This procedure is not only simple but gives plugins a lot of freedom.

The communication, therefore, is done line by line. Since `plugger` runs everything in the background, there is no delay between plugin executions. If a plugin is slow — such as querying Wikipedia — and was executed first than one that is fast, then users would not have to wait. Also, `plugger` will handle all the logic necessary for flood — though this has not yet been done. That is, anything that is spoken by a plugin is saved in a correspondent plugin buffer which is passed back to Dansarina on a slow enough rate; for example: one line per two seconds. This decision avoids Dansarina from possibly getting in trouble with IRC servers, and this decision puts the burden of the work on the plugin daemon, making Dansarina free of the complications of the required algorithms.

2.1 The triggering of events

Dansarina will be delegating tasks to plugins through `plugger` upon events. For example, if someone changes a topic, dansarina receives a `TOPIC` message which in turn triggers her to run any plugin whose file name starts with `ontopic::` — that is, plugin writers may write plugins that will be automatically executed as particular events are generated. This makes Dansarina flexible enough to implement features. Notice the tense of the verbs in this paragraph; these have not yet been implemented, but it's very easy to implement support for these events.

For example, it's easy to implement a topic keeper for an infinite number of channels, or a topic history plugin; it's easy to implement a "last seen" plugin which informs when a user was last seen in any channel Dansarina has been watching; store the quit messages from users, for fun. It would be easy to handle nickserv or chanserv messages, et cetera.

These tasks would be easily implemented by the AWK programming language; the ones that need the network would be helped by the `tcpclient` program. This infrastructure could also make Dansarina feasible to live in a `chroot` jail.

2.2 The implementation of `plugger`

Since `plugger` works like a shell, we're going to have a main loop in which it waits for input, which would always come from dansarina. Also, we need some structure like a `peer struct` to keep track of all plugins currently running. Each of them would have their own buffer; for each plugin, we need only one buffer because all the information we pass is only through command line arguments. But the `peer` structure is too particular for dansarina. To keep things independent from each other, I'm going to build a similar structure called `plugin`.

Notice that `plugger` listens from the standard input and talks to the standard output; dansarina will be responsible to redirect file descriptors connecting her pipes to the standard input and output of `plugger`. So, the main loop will be always listening for input from the standard input.

2.2.1 The main loop

Things that must be polled for are: (1) output from plugins, (2) action in the standard input, (3) signals from children. To avoid zombie processes, and knowing that we will always have many children running at the same time, we will be catching `SIGCHLD` signals and running `waitpid` with the option `WNOHANG`; this takes care of (3). For (1) and (2), we will call `poll` to let us know of activity in the file descriptors. So, the main loop is a procedure that calls `poll` and takes an action when `poll` returns.

We will be calling `waitpid` because we want to wait for any children that may be terminating. We will be running them in random order, and we must catch the first one that terminates, so that we can get the information about their exit status from the kernel telling it to throw the information away; this keeps us from generating zombie processes. The argument `-1` for the parameter `pid` of `waitpid` gives us this solution.

5 ⟨*plugger.c* 5⟩≡

```
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>
#include <dirent.h>
#include "strerr.h"
#include "io.h"
#include "poll.h"
#include "byte.h"
#include "sig.h"
#include "getln.h"
#include "scan.h"
#include "min.h"
#include "str.h"
#include "plugger.h"

#include <stdio.h>

extern void free(); extern void* malloc();

int poll_max; struct pollfd fd[MAX]; struct plugin proc[MAX];

static void log(char *s)
{
  FILE *log;
  log = fopen("./plugger.log", "a");
  fputs(s, log);
  fclose(log);
}

static void sigchild(int sig)
```

```
{
  pid_t pid; int stat;

  for (;;) {
    pid = waitpid(-1, &stat, WNOHANG); if (pid <= 0) break;
  }
}

int main()
{
  int r; int i; int pfd[2]; int nsocks; io in; char path[1024];
  char bf[512*2]; int len; char **argv[7]; char ln[512];

  sig_catch(SIGCHLD, sigchild);

  for (i = 0; i < MAX; ++i) {
    proc[i].poll = &fd[i]; plugin_zero(&proc[i]);
  }

  plugin_attach(0); /* attach standard input: dansarina */

  for (;;) {
    nsocks = polltimeout(fd, poll_max + 1, 120);

    if (nsocks == 0) { continue; }

    if (fd[0].revents & (POLLIN | POLLERR)) {
```

This definition is continued in chunks 6–12.
Root chunk (not used in this document).
Defines:
  `log`, used in chunks 6, 8, 11, and 12.
  `main`, never used.
  `poll_max`, used in chunks 7 and 12.
  `sigchild`, never used.
Uses `plugin_attach` 12 and `plugin_zero` 12.

This point is reached when dansarina is speaking with `plugger`, and she's probably making requests for more executions. The first thing we do is to check if dansarina makes any sense. We will validate her sentences because we don't trust her. Also, notice that she may be fast enough to make two or more requests at once, so in the call to `getln`, we will be getting all input available in `plugger`'s standard input. So, we loop through all lines and process all requests one by one. If her request was not resonable, we should silently ignore.

6 ⟨*plugger.c* 5⟩+≡

```
    io_set(&in, read, 0, bf, sizeof bf);

    /* do { */
```

```
    len = getln(&in, ln, '\n', sizeof ln);

    if (len == -1) {
       log("quitting. getln() -1\n"); _exit(1);
      /* continue; */
    }

    if (len == 0) {
       log("quitting. getln() 0\n"); _exit(1);
      /* continue; */
    }

    ln[len] = 0;

    log("from dansarina: "); log(ln);

    r = plugin_argv(ln, argv, sizeof argv);
    if (r) {
      log("plugin_argv failed\n");
      /* continue;  */
    }

    r = plugin_exist(argv[0]);
    if (!r) { log("plugin doesnt exist\n"); /* continue; */ }

    r = plugin_path(argv[0], path, sizeof path);
    if (r) { log("plugin path error"); /* continue; */ }

    r = plugin_exec(path, argv);
    if (r) { log("could not exec"); /* continue; */ }

  /* } while (in.p); */

  if (--nsocks <= 0) continue;
}
```

Uses log 5, plugin_argv 9, plugin_exec 11, plugin_exist 10a, and plugin_path 10b.

This is it. Now, if any children said anything, poll will wake us up, and we will be noticing who spoke in the loop below. Notice that we only go up to poll_max to avoid losing time. What we never do, however, is decrease poll_max when the highest plugin is terminated, but we should.

7 ⟨*plugger.c* 5⟩+≡

```
for (i=1; i <= poll_max; ++i)
  if (fd[i].revents & (POLLIN | POLLERR))
    plugin_doit(&proc[i]);
```

```
    if (--nsocks <= 0) { continue; }
  }
}
```

Uses plugin_doit 8 and poll_max 5.

2.2.2 Reading from plugins

The function plugin_doit has a simple job to do: read every line printed by the plugins. The parameter that is receives is a pointer to a struct proc. With this pointer, we have the plugin with us; we have its descriptors, its buffers, et cetera. We can take care of it entirely. For example, if the plugin dies unexpectedly, we will probably get an error from a read call, which in turn we notify dansarina and clean up by closing and freeing resources allocated for that plugin. If the plugin exits successfully, we do similarly: we pass the information back to dansarina, and free up the allocated resources.

8 ⟨*plugger.c* 5⟩+≡

```
void plugin_doit(struct plugin *p)
{
  char ln[512]; int len;

  /* do { */
    len = getln(&p->in, ln, '\n', sizeof ln);

    if (len == -1) { /* log error? */
      log("plugin_doit: getln -1\n");
      plugin_detach(p); /* break; */
      return;
    }

    if (len == 0) { /* log success? */
      log("plugin_doit: getln 0\n");
      plugin_detach(p); /* break; */
      return;
    }

    ln[len] = 0; log("I read this: "); log(ln);

    outsflu(ln);

  /* } while (p->in.p); */ /* buffer =/= empty */
}
```

Defines:
  plugin_doit, used in chunk 7.
Uses log 5 and plugin_detach 12.

It's worthwhile to remember that as the children die, the kernel will deliver a signal to `plugger` so, there's nothing required for us to do here. We need only free up the resources, and quit.

2.2.3 Building `argv`

The next two functions build the `argv` array and execute the plugin. It's a tedious task, but straightforward: we read each word in a dynamic allocated `argv` array, and for the last argument, we scan the rest of the line; finally, we close the array. To do that, we use `scan_word` to count how long the argument is, and we use the information to allocate memory. If the argument fails to match the maximum allowed length, then we will truncate it and skip the part that didn't fit. This will make it easy for plugins and users to notice when they're exceeding the maximum length for each argument — the limit is per argument, not per line. We scan four arguments; the fifth is everything up to the end of the line — not exceeding 256 bytes.

9 ⟨*plugger.c* 5⟩+≡

```
int plugin_argv(char *ln, char **argv, unsigned int size)
{
  char *s; unsigned int n; int i;

  s = ln;

  for (i = 0; i <= 4; ++i) {

    if (!*s) return -1;

    n = scan_word(s, min(32, str0_len(s)), s); if (!n) return -1;

    argv[i] = malloc(n+1); if (!argv[i]) return -1;
    byte_copy(argv[i], n, s); argv[i][n] = 0;

    /* if it exceeds, advance until next word */
    if (n == 32) { n = scan_word(s, str0_len(s), s); } s += n + 1;
  }

  n = scan_line(s, min(256, str0_len(s)), s);
  if (!n) { argv[i] = 0; return 0; }

  /* two extra bytes: one for newline, one for 0-terminator */
  argv[i] = malloc(n+2); if (!argv[i]) return -1;
  byte_copy(argv[i], n+1, s); argv[i][n+2] = 0; s += n + 1;

  argv[i+1] = 0; return 0;
}
```

Defines:
plugin_argv, used in chunk 6.

2.2.4 Checking for plugin existence

Open directory, scan for plugin program; return 1 if found, 0 otherwise.

10a ⟨*plugger.c* 5⟩+≡

```
int plugin_exist(char *s)
{
  DIR *u; struct dirent *e; unsigned int n;
  u = opendir("./run/"); if (!u) strerr_sys(1);

  for (;;) {
    e = readdir(u); if (!e) return 0;

    if (e->d_name[0] == '.') continue;
    n = str0_len(e->d_name); if (n != str0_len(s)) continue;

    if (str0_cmp(e->d_name, n, s))
      return 1;
  }
}
```

Defines:
plugin_exist, used in chunk 6.

2.2.5 Building the path

Not only building the absolute path of the plugin is required, but we must also verify that it exists. We will use `getcwd` to get the current directory; then we append the plugin directory, and then the plugin name.

10b ⟨*plugger.c* 5⟩+≡

```
int plugin_path(char *s, char *path, unsigned int size)
{
  unsigned int n; char file[16]; static str p;

  p.bf = path; p.n = size;
  n = scan_word(file, min(16,str0_len(s)), s); file[n] = 0;

  getcwd(path, size);
  if (str0_len(path) + str0_len("/run/") >= size - n) return -1;

  p.p = str0_len(path); p.n -= p.p;
```

```
  if (!str_PUTS(&p, "/run/")) return -1;
  if (!str_PUTS(&p, file)) return -1; str_0(&p);

  return 0;
}
```

Defines:
plugin_path, used in chunk 6.

2.2.6 Executing the plugin with `execve`

To execute a plugin, we must attach it to `proc`, prepare the pipes, `fork` and `execve` the plugin. Since we're attaching the plugin to the `proc` array, we will be handling any activity from the plugin once `poll` tells us so.

The function `pipe` creates one pipe, with a pair of file descriptors. After creating the pipe, we will `fork`, and so we will have two pipes plus four file descriptors. From these four, two will be closed: one will be closed by `plugger` and another will be closed by the `plugger`'s child — which soon enough will be replaced by a plugin. We must save the two file descriptors that will stay open.

Plugins don't care about descriptors, they get input only from the command line and print their lines to the standard output. It is `plugger` that must save information such as file descriptor numbers. To save this information, we use the plugin space in the array `proc`. Since each element contains a pointer to a `struct poll`, we save this descriptor right there. We will read from it. We may forget about the output descriptor once we set up the child's standard output.

11 ⟨*plugger.c* 5⟩+≡

```
int plugin_exec(char *path, char **argv)
{
  int r; int fd[2];

  if (!plugin_emptyslot()) {
    log("No empty slot.\n");
    return 0;
  }

  r = pipe(fd);
  if (r < 0) { return -1; }

  r = fork();
  if (r < 0) { return -1; }

  else if (r > 0) {
    plugin_attach(fd[0]); close(fd[1]); return 0;
```

```
  }

  else {
    close(fd[0]);

    if (fd[1] != 1) {
      if (dup2(fd[1], 1) != 1) exit(1);
      close(fd[1]); /* we can close the duplicate */
    }

    execve(path, argv, (void *) 0); _exit(4);
  }
}
```

Defines:
  `plugin_exec`, used in chunk 6.
Uses `log` 5, `plugin_attach` 12, and `plugin_emptyslot` 12.

Notice what we have done: we created the pipes, and we forked. The parent closed one of the descriptors, but attached the other to its plugin array with a call to `plugin_attach`. This means that we will be polling for activity on that descriptor. That's all the parent does: closes a descriptor and attaches another. Now, the child closes its input descriptor because it won't be reading anything from the parent. It duplicates its output to match the descriptor `1`: the standard output of the plugin. This means that the plugins will be able to print to their standard output and be read by `plugger`. Now, after duplicating the output descriptor, we have a duplication evidently, so we may close the descriptor that we don't need anymore. If `execve` fails, we `exit(4)`.

3 Other stuff

Below you find other functions used in this program; `plugger` is almost entirely in these pages; what you don't find here are library functions. I could have made a little library for `plugger` with the `plugin_*` functions; if it grows much bigger, we may do something like that.

12 ⟨*plugger.c* 5⟩+≡

```
void plugin_zero(struct plugin *p)
{
  byte_zero(&p->bf[0], sizeof p->bf);
  io_set(&p->in, read, -1, &p->bf[0], sizeof p->bf);
  p->poll->fd = -1; p->poll->events = 0; p->poll->revents = 0;
}

void plugin_flood(int s)
{
  /* log("plugger: sorry, too many plugins running.\n"); */
```

```
}

void plugin_set(struct plugin *p, int fd)
{
  io_set(&p->in, read, fd, &p->bf[0], sizeof p->bf);
}

int plugin_emptyslot()
{
  int j; struct plugin *p;

  for (j = 0; j < MAX; ++j) {
    p = &proc[j]; if (p->poll->fd == -1) return 1;
  }

  return 0;
}

struct plugin * plugin_attach(int s)
{
  int j; struct plugin *p;

  for (j=0; j < MAX; ++j) {
    p = &proc[j];
    if (p->poll->fd == -1) {
      p->poll->fd = s; p->poll->events = POLLIN; plugin_set(p, s);
      break;
    }
  }

  if (j > poll_max) poll_max = j;

  return p;
}

void plugin_detach(struct plugin *p)
{
  close(p->poll->fd); plugin_zero(p);
}
```

Defines:
  plugin_attach, used in chunks 5 and 11.
  plugin_detach, used in chunk 8.
  plugin_emptyslot, used in chunk 11.
  plugin_flood, never used.
  plugin_set, never used.
  plugin_zero, used in chunk 5.
Uses log 5 and poll_max 5.

4 The protocol

Dansarina request a plugin execution by printing a request command in the standard input of `plugger`, which expects the line in the following format

```
pluginname nickname username hostname channel p1 p2 ... pn
```

So, although we could, we won't allow some characteres in plugin names; for example, space is not valid. It will follow from our code, however, that tabs are, new lines are, and every other UNIX valid character for a file name, but we will discourage that — how would IRC clients call plugins that contain a `^G` in the name? As soon as `plugger` reads those lines, it will load the plugin in the background and wait for its input.

4.1 A `chroot` cage?

Could we benefit from a `chroot` jail? I'm not sure if it is interesting to do that, but I'm lately thinking about it because of my new thoughts on the plugin system. Lately — `Sat Aug 05 14:51:07 EDT 2006` — I have been bringing up the idea of allowing one kind of plugin: scripts `awk`. It turns out that AWK is this beautiful programming language that allied with a `tcpclient` program could give us all the resources that we need, with few penalties, and still make dansarina extra safe by running her in a `chroot` jail. Is this an interesting step? I think so.

4.2 On the independence of the programs

Dansarina is dependent only on the `libc`, and uses very little of it. Statically linked, dansarina takes 230 kibibytes. A dynamic linked `awk` takes about 140 kibibytes, so I don't think it would be too big when statically linked because it depends on the `libc`, `libm` and `libgnuregex` — of these, only the `libc` is big, but is small enough for me to believe that when I link `awk` statically against it, we'll get a feasible size.