
debconf6
proceedings
hot & spicy
http://debconf6.debconf.org
Oaxtepec, Mexico, May 14th - May 22nd 2006

Contents

Part I.

Talks

1. Releasing in Time - Etch in December 06

by Andi Barth and Steve Langasek

Abstract

Debian makes a ”stable release” from time to time. This article shows how the background work looks like, what the major issues are and which improvements are planned; the intended audience are people knowing Debian closely. The authors are the Debian Release Managers. More information is available at `http://release.debian.org/`

Releasing means

Sarge, the recent stable release of Debian, was published at beginning of June 2005 after about three years of hard development. The packages included in that release began their life in ”unstable” (border cases are discussed later on). If they were fit enough, they were automatically copied to testing; ”fit enough” is determined by a script called britney that pushes packages from unstable to testing that survived a minimum time (usually ten days), are not too broken themselves and do not break other packages in testing. Also, for packages not meant for the next stable release, there exists a suite called ”experimental”; packages in that suite stay where they are and provide a playground for changes.

The task of release management in Debian is to make sure that Debian can actually release. This sounds like a trivial statement. But in fact, that is our standard for all decisions. It is a hard fight to make sure that testing does not become more buggy. Also, a lot of other people need to be kept in the loop. Debian can only release if almost all people working on Debian believe it. That includes speaking with all the major teams (ftp-masters, installer, tool chain, (base) package maintainers, X, and so on).

Behind the curtain

Our most famous tool is ”britney”, the release team’s working tool that updates testing from unstable. Other tools include the lists of release critical bugs, both at `http://bugs.debian.org/release-critical` and `http://bts.turmzimmer.net/details.php`. Also, there are scripts to track the security status on `http://merkel.debian.org/~joeyh/testing-security.html`. Of course, also the ”normal” tools like madison, grep-dctrl and debdiff are used very much (and partly with small helper scripts).

Britney starts considering packages after 10, 5 or 2 days, depending on how urgent the uploads since the last testing migration are. A package needs to be in sync on all architectures, and also be less buggy than the package currently in testing. Also, they must not break another package in testing. Britney is a python script, mixed with c-code, and available at `http://ftp-master.debian.org/testing/update_out_code/`

Another way to get a package in testing is via testing-proposed-updates; that requires a release team member to manually approve the package. For sanity reasons (as the new package is more or less untested), it is preferred that packages go in via unstable. (Technically, also this part is taken care of by britney.)

Britney is run shortly after the daily install on ftp-master, and uses the packages lists of testing and unstable as starting point. Britney produces a file suitable to synchronise the database with by heidi, an ftp-master’s tool; the updated database can however be seen only after the next day’s installer run. Also, britney produces the so called ”testing excuses” and ”testing output” that indicates why a package did (not) make it to testing. These explanations are deciphered on `http://bjorn.haxx.se/debian/`.

The release team can influence the result of britney by usage of hints. They especially allow to remove (buggy) packages, freeze and thaw packages and ignore some of the usual preconditions for a testing migration. Britney can be re-run during the day if the needs arise.

Release standards

A package can only be part of the next stable release if it doesn’t contain critical nor grave bugs (makes the package or the system unusable or introduces a security hole). It needs also to be releasable from the maintainers point of view,

and it must meet the release standards. The canonical list of release standards is available from `http://release.debian.org/`.

Why managing

Woody has 11 different kernel source packages. Sarge had at one time 17 different kernel source packages. Together with the kernel team, the release team was able to reduce the number of source packages to three, one per 2.2, 2.4 and 2.6 kernel major releases. This is quite important to be able to make security support. It also made the life of the installer team easier. However, that showed also an quite important problem - there are currently no maintainers for two architectures, which meant round trip times of about two months for security updates via unstable.

Heading towards etch

As of writing this, we are more than halfway through to release of etch. Working constantly towards etch has paid itself. We have a quite good working toolchain, a fair overview about the open issues, and a limited number of RC bugs. Etch should release beginning of December 2006.

To make this target date realistic, the release team minimised the number of release blockers. A lot more wishes became ”pet release goals” which means they will not be allowed to block etch, though it would be really nice to get them also be done in time for etch.

Our list of release blockers contained the gcc-4.0 transition and the xorg transition, which are both done. Amd64 is now part of unstable and will become part of testing in time. Sorting out the documentation in main is slowly going on, but it is happening - with the recent General Resolution on the GFDL-documentation, also all open questions are settled for that. Some questions remain open with secure apt, but it is generally working. All other wishes (as nice and useful they might be) were decided to not be release blockers.

Architecture status

For the first time, a formal requirement for release architectures was defined. We noticed quite fast that the requirements alone improved handling of the architectures where we had issues with, and it also showed where the issues were basically only on the top, or if the problems are in reality too large for keeping the architecture in the next stable release.

The basic core of the architecture requirements are: Does the architecture add more value to Debian, or is it rather only a source of pain? If it’s the first, it is welcome. If it’s the second, we can only afford a certain amount of pain. On this high-level, the necesserity of architecture requirements are undoubted. However, in going more into detail, there were some discussions. One has to draw a decision line somewhere - for example, whether 50 users are required, and not 40 or 60 is a wilfull decision. That one has to put up some requirement on having real users is natural and correct.

In the end, we think our requirements were pretty sane. We found out that most architectures managed to go through easy enough, and the remaining architectures have severe issues like no debian-installer.

Timeline

Soon, the first parts of debian will be frozen - at the beginning of July the toolchain, the kernels and base will be the first ones. Soon after that, the final release candidate of the debian installer will be done, and the installer will be frozen. Two months later, the general freeze will set in, so that we can release beginning of December.

2. Debian Community Guidelines

by Enrico Zini

Abstract

This is the *Debian Community Guidelines*, a collection of what sane people are already doing everyday, written down so that we don't risk forgetting.

The Guidelines are divided in four parts:

- main suggestions to always keep in mind.
- suggestions specific to communicating with others.
- suggestions specific to working with Debian.

These guidelines are mainly about interacting with other people. For technical suggestions, please refer to the Debian Developer's Reference[1].

In addition to the guidelines, there is a collection of resources for moderators, listmasters, DPLs and other people with some leadership, moderation, mediation or facilitation role (starting on page 13).

Main guidelines

- *Strive for quality*

  Everything you post will contribute to the knowledge, the usefulness and the look of Debian. It will be publicly archived and found by others over time and in different contexts. Make informed comments, keep the information level high and the stress level low.

- *Work with others*

  Try to improve the quality of the community. Co-maintain packages. Share your ideas and plans. Allow people to help and contribute. Thank people.

  If you disagree with someone, explain your reasons. Listen to the reasons of others. Some disagreements can be lived with as long as people can still work together. Spend time working on a convincing result rather than on convincing others.

- *Principles do not change, the rest changes with work*

  Some things are unchangeable: social contract, DFSG. The rest you can completely change with your work.

- *Support the reasonable, rather than attack the arrogant*

  When someone is behaving badly, supporting the victim of the bad behaviour helps solving the situation much more than attacking the misbehaving person.

  It helps both in undoing the results of the bad behaviour and in showing what is a better way of doing things.

  Attacking the misbehaving person would just add tension, yet leave the victims with their problems unsolved.

  For example, if one got a bad reply with a rude RTFM or a rant, you can post a message with the real solution, some useful suggestions, or a more precise pointer to specific documentation. This will be helpful to the original poster, it will be useful information for the list archive and will serve as an example of a better way to reply.

[1] `http://www.debian.org/doc/developers-reference`

Communication-specific guidelines

Introduction

One of the big problems of large communities is to keep communication effective, and the challenge is to keep providing quality and compact information to everyone.

This becomes difficult when many people participate in a discussion often, as the amount of text and time spent in writing grow more quickly than the amount of information that is actually produced.

It is important to keep focused on improving our body of knowledge, just like we keep improving our body of software: working online, messages keep living after their recipient has read them, and become public pieces of information that can show in the list archives[2] or the Debian BTS[3], get indexed in search engines, linked by other people, quoted or pasted in an HOWTO or an FAQ and so on.

Improving means caring for the information to have quality content and to be communicated efficiently. It also means making the process of producing and improving of our body of knowledge pleasant and sustainable over time.

What follows is a collection of suggestions about these three aspects, plus practical suggestions to cope with recurring communication problems.

Improving the content

The main part of communication is content, and the quality of the content is the main element that makes it useful.

If the quality of your posts is high, you go a great length in reaching the goal you had when writing your message. If the quality of your posts is usually low, people will slowly start skipping your messages, even if (and especially if) you post very often.

- *Ensure you are adding useful information*

  Make sure that every one of your posts contributes at least a piece of missing useful information.

  For example, ”RTFM” without a precise pointer is not useful information: if someone already knew where to RTFM, they wouldn't bother to ask.

  Try also not to repeat your points: when you have a good argument, idea or claim, let it be heard once. Do not insist with it unless you can come up with code, specifications or technical explanations to back it up.

- *Share the help you receive*

  Solving a problem creates knowledge, and it is important to share it. A useful IRC or email conversation can become a blog entry, a wiki page, a short tutorial or HOWTO.

  When you receive help, try to take notes of all it actually takes you to completely solve the problem, and share them.

- *Reuse existing resources*

  When you are asking a question, *do some research before posting*: not only finding something on Google will give you an immediate answer, but it will avoid making people upset by asking the same question over and over again.

  Often you may end up reading a list archive with mails related to your problems. Follow the threads to learn more from the experiences of other people.

  If you see an error message you don't understand, pasting it in the Google search field often gives you useful information.

  If you are replying to a question instead, a bit of research will allow you to provide a more precise answer.

  When you find useful informations, *post pointers to existing resources*.

  If a problem has already been solved or a point has already been made, post a link to the existing resource rather than repeating previous arguments.

  If the existing resource is not good enough, try to improve it: either in place (for example, if it is in a wiki) or by posting your further comments together with the link.

- *Know what you want* and *make sure people know what they want*

  The biggest part in getting a good answer is to make a good question: try to work out what is that you really want. Ask yourself what are you really trying to get, where is that you are stuck, what were you expecting that

[2] `http://lists.debian.org`
[3] `http://bugs.debian.org`

did not happen. If you are lost, try asking yourself these four questions (this is called ”Flanagan’s Critical Incident Technique”):

- What led up to the situation?
- What did you do that was especially effective or ineffective?
- What was the outcome or result of this action?
- Why was this action effective, or what more effective action might have been expected?

The same questions are also useful to help clarify a situation when the other person is confused, and can be very handy when replying to a request for help or a bug report.

Improving the presentation

The way you present your content is also important. There is no absolute ”most appropriate” way to present a content, but one has to choose what is better depending on who are the readers and what is the goal of the discussion.

In the case of technical discussion, what most people want is to quickly and efficiently find the best solution to a problem. Follows a list of style suggestions for contributing to a technical discussion:

- *Put the main point at the beginning, and the long details later.*

  If people have to read through lots of text in order to get to the main point, they are likely to just skip the message if they are busy.

  Hook the readers in by stating the main point right away instead of trying to create expectation and surprise.

  You can even avoid posting about the long details and put them on the Debian wiki[4] instead.

- *Talk with code or patches.*

  Wherever it is possible to use them, code or patches are the most efficient way of conveying technical information, and can also be used, applied or tested directly.

  Often patches are the preferred writing style: technical people tend to like to read code more than they like to read English, except when English is written inside code comments.

  Writing code is also useful to clear your own mind: while writing a piece of example code to show a problem, it is quite common to find the solution or to reframe the problem in a clearer way.

  Finally, some things are harder to explain in English rather than with code: so if there is something you have problems saying in English, try to say it using your favourite programming language.

- *Start new threads when asking new questions.*

  In Debian, most people are taking advantage of the ”threading” feature of their mail programs. This means that it is important for them that messages in a thread are connected with the main topic discussed in the thread.

  If you want to post a message which is not a contribution to an existing thread, please create a new message to start a new thread instead of doing a reply. This will show your message properly, and will not disrupt other discussions.

- *Point to existing resources.*

  If a problem has already been solved or a point has already been made, you can greatly condense your message by just posting a link to it.

  If the existing resource is not good enough, try to improve it: either in place (for example, if it is in a wiki) or by posting your further comments together with the link.

- *Use a plain and simple style*

  People should spend their time answering your question or using your information, rather than understanding it.

  To help it, try to keep formalities to a minimum and avoid confusing people with rethorics:

  - ask questions without asking if you can ask or apoligising for asking. This may be impolite in real life, but it is generally the preferred way online.

[4] `http://wiki.debian.org`

  - if you are not a native English speaker, you don't need to apologise for it: a minimal knowledge of the language is enough, as long as your mind is clear and you use a plain style.

    If English makes it really hard for you, you can look for a Debian list where they speak your language. Many of them are available in the list of FOO[5].
  - Only ask a question if you want to know the answer: rethorical questions often come across as unnecessarily confrontational, so avoid them if you can. If you have an opinion, just state it and ask for correction if needed.

  There is of course place for advanced uses of language and style in Debian: you can indulge in creative writing in places like the debian-curiosa mailing list[6], or in your blog that you can then have syndicated on Planet Debian[7].

Ensuring sustainability

While good messages are important, ensuring that the project keeps being a productive and fun place to be requires some care towards relationships with other people. This mainly boils down to being smart, honest and polite.

Most of the following suggestions are likely to be trivial for everyone, but they are worth collecting anyway so that we avoid forgetting some of them.

- *Read messages smartly.*

  Most of the fun with working with others depends on how you read and interpret their messages and contributions:

  - *Exercise your will.*

    It is finally up to you to decide how a message is important and how it influences the way you do things.

    A message is ultimately important if it meets your needs and motivations: other aspects of it, such as if it is the last message in a thread or if it is full of convincing rethorics or catchy phrases, have no relevance in a technical discussion.
  - *Interact with human beings instead of single messages.*

    When you see a message you don't like, try to remember your general opinion of the person who wrote it. Try not to make a case about a single episode.

    If you see a message you don't like written by a person you don't know, wait to see some more posts before making assumptions about the person.

    If people do something that seriously undermines your trust or opinion of them, try writing them privately to tell about your disappointment and ask about their reasons.
  - *Be forgiving.*

    Try to be tolerant towards others; when you are unsure about the intentions of a message, assume good intentions.
  - *Be tolerant of personal differences.*

    Remember that you might be discussing with someone who normally looks, thinks or behaves very differently than you do.

    Debian is a large project which involves people from different cultures and with different beliefs. Some of these beliefs are understood to be in open conflict, but people still manage to have a fruitful technical cooperation.

    It is normal to dislike someone's beliefs but still to appreciate their technical work. It would be a loss if a good technical work is not appreciated because of the beliefs of its contributor.

- *Be positive before being negative*

  Try to put positive phrases before negative phrases: the result is more rewarding and pleasant to read.

  This can make a difference between a bug report that looks like an insult and a bug reports that motivates developers to carry on their work.

  It is an interesting and at times difficult challenge to do it: you can look at how these guidelines are written to see examples.

  One reason this is hard to do is because most of the work with software is about problems. For example, most of the conversations in the Bug Tracking System[8] start with telling that something does not work, or does not work as expected.

[5] `http://lists.debian.org/`
[6] `http://lists.debian.org/debian-curiosa`
[7] `http://planet.debian.org`
[8] `http://bugs.debian.org`

This naturally leads us to mainly talk about problems, forgetting to tell us when things work, or when we appreciate the work of someone.

- *Give credit where credit is due.*

  Always acknowledge useful feedback or patches in changelogs, documentation or other publicly visible places.

  Even if you personally do not care about attribution of work you've done, this may be very different for the other person you're communicating with.

  Debian is an effort most people pursue in their free time, and getting an acknowledgement is often a nice reward, or an encouragement for people to get more involved.

- *Be humble and polite.*

  If you write in an unpleasant manner, people won't feel motivated to work with you.

- *Help the public knowledge evolve.*

  - *Reply to the list.*

    If you can help, do it in public: this will allow more people to benefit from the help, and to build on top of your help. For example they can add extra information to it, or use parts of your message to build an FAQ.

  - *Encourage people to share the help they receive.*

    Solving a problem creates knowledge, and it is important to share it. A useful IRC or email conversation can become a blog entry, a wiki page, a short tutorial or HOWTO.

    When answering a question, you can ask the person to take notes about what it takes to actually completely solve the problem, and share them.

  - *Sustaining a discussion towards solving a problem is sometimes more important than solving the problem.*

    The most important thing in a technical discussion is that it keeps going towards a solution.

    Sometimes we see a discussion and we can foresee a solution, but we do not have the time to sort out the details.

    In such a case, there is a tendency to postpone answering until when one has the time to create a proper answer with the optimal solution in it.

    The risk is that the time never comes, or we get carried away by other things, and everything we had in mind gets lost.

    When there is this risk, keeping the discussion going towards solving the problem is more important than working silently towards the solution.

    See a post on Enrico Zini's blog[9] for a personal example.

Communication mini-HOWTOs

Bringing long threads to a conclusion Long threads with points being repeated over and over are an annoying waste of time. They however tend to happen from time to time, especially about important and controversial topics where a consensus is not easy to reach.

This is a collection of suggestions for coping with it:

- *Discuss controversial points off list with the people you disagree with.*

  This would make it easier for you both to reach some consensus which is satisfactory for both.

- *Take a leadership role*

  Step forward, take the good ideas that were spoken in the thread and work to put them into practice.

  This is good if good ideas have been spoken but the discussion keeps going with nothing else coming out, or in case of a thread that tends to be recurring over time.

  You can also put together a group of interested people to work on the issue. You can do it by taking the lead, contacting one by one the other active people in the thread to see if they are interested, making a public announcement to let people know and ask if more want to join.

[9] `http://www.enricozini.org/blog/eng/converging.html`

- *Make summaries*

  Summarise important parts of the thread and make them available to everyone, so that they don't have to be repeated again.

  This can be done for recurring points, or if the thread reaches a useful point, or a consensus.

  The summary can be done on the Debian Wiki[10]. For an example, have a look at the collection of ideas to improve the Debian release schedule[11].

- *Start a new thread when the discussion drifts away*

  Sometimes the discussion in a thread drifts away from the original topic to something related, or even unrelated.

  When this happens, do your best to reply starting a new thread.

It is also important to learn to read long thread: you can find many useful tips for it on an article in Joey Hess blog[12].

Coping with flamewars Stopping a flamewar is hard, but one can try to slow it down. A good way is to send a private mail to some of the most heated posters, such as:

```
This is off-list.

What I think of the situation is [short summary].

However, this discussion is not being useful anymore: please drop the
thread.

Best wishes,
```

This will tell them that the thread is becoming annoying, and also offer them a chance to continue with the discussion off-list.

Posting a public message asking people to stop the thread does not work, and usually creates even more unwanted traffic.

Another useful way of using private messages is to contact friends if they are being unconstructive or flameish. This would be better done when you otherwise agree with their point, so that the remark can be fully about how they are saying things rather than about what they are saying.

Debian-specific guidelines

Package management

Many people in Debian have found it useful to have packages maintained by groups rather than by singles. Here is a list of ways to do it:

- Work with comaintainers. This works particularly well when a group of people with similar interests share maintainance of a related group of packages. For example see pkg-ocaml-maint[13], pkg-italian[14] and pkg-games[15], but it also work well for single packages, like pkg-vim[16].

  Alioth[17] comes particularly useful when doing group maintainance, as it easily allows to share a VCS repository, to create mailing lists and to also include in the maintainance people who are not yet Debian Developers.

- Using a distributed revision control system like Bazaar-NG[18] or Darcs[19], you can put a mirror of your tree on your webspace at people.debian.org[20], and engage in cross-pulling of patches with other interested people.

[10] http://wiki.debian.org
[11] http://wiki.debian.org/ReleaseProposals
[12] http://www.kitenet.net/~joey/blog/entry/thread_patterns.html
[13] http://pkg-ocaml-maint.alioth.debian.org/
[14] http://alioth.debian.org/projects/pkg-italian/
[15] http://wiki.debian.org/Games/Development
[16] http://pkg-vim.alioth.debian.org/
[17] http://alioth.debian.org
[18] http://bazaar-vcs.org/
[19] http://darcs.net/
[20] http://people.debian.org

This allows an easy start, with you normally doing your work and having it exported to a website, and is open to scaling up anytime more people want to get involved.

If you are not a Debian Developer, and thus cannot use the webspace at `http://people.debian.org`, you can create an account on Alioth[21] and use the webspace it provides. Note that webspace on Alioth is visible as `http://haydn.debian.org/~user/`.

- Add yourself to the low threshold NMU list[22].

  This lets fellow Debian Developers know that you are open to have external contributions in your packages.

Handling bug reports

- *Take bugs positively.*

  Bug reports are precious information that allows you to improve the quality of your packages: they are not insults to your skills, and there is nothing shameful in having a bug reported to your packages.

  Having bugs reported is actually a sign that people are using your packages and are interested in them.

- *Interface with upstream.*

  As the maintainer of a package, you are also an interface between Debian users and upstream developers.

  This can mean tracking upstream's development mailing lists, filtering bugs reports in the Debian BTS and forwarding relevant ones to upstream's bug tracking system. It is also needed to track what happens in upstream's bug tracking system with regards to bugs that are also reported to Debian.

  If this means a lot of work, which is usually the case for popular packages, it can be a good idea to involve more people to help with the task. Usually the more a package is popular the more bugs are reported, but it is also easier to find people that could help.

- *Be open minded.*

  Bug reporters could be different people than what you expect: try not to make assuptions about them.

  People could also be using the software in a different way than you do, and it may happen that your proposed solution to a problem might not work well for them, and a more general solution is needed.

  These are four questions that can be very useful when the reporter seems to be working differently than what the developer expects:

  - What led up to the situation?
  - What did you do that was especially effective or ineffective?
  - What was the outcome or result of this action?
  - Why was this action effective, or what more effective action might have been expected?

Reporting bugs

Reporting a bug is an important way of helping with development: when reporting a bug it's then very important to be helpful.

While this may seem obvious, it can turn out to be difficult to do when a bug is reported after the frustration of struggling with a problem.

It is important to be in a good mood when reporting a bug. It can help to take a breath or a walk to the fridge (unless it's empty) to allow frustration to wear off.

If the bug comes from a trivial mistake, try to remember of the trivial mistakes you did in the past: don't necessarily assume that the maintainer knows less than you.

If you find trivial mistakes, however, the best thing to do is to make sure that the main point in the report is a nice patch to fix them.

Finally, other poeple may not use the software in the same way that you do: this means that, if you propose an idea for solving the problem, you should be prepared for it not to be accepted if there is a more general solution.

[21] `http://alioth.debian.org`
[22] `http://wiki.debian.org/LowThresholdNmu`

Resources for moderators, listmasters, DPLs...

This section contains a list of mostly theoretical resources that could be useful for moderators, listmasters, DPLs and other people who might be in need of extra resource to handle a conflictual situation.

It presents a list of concepts that help in having a better understanding of the situation and the possible ways out.

Note: the contents still sound academical and could use being rephrased in an easier language. Help is needed with it.

Conflict

Conflict is a substantial element in our social life.

> Social conflict is an interaction between actors (individuals, groups, organizations, etc), in which at least one actor perceives an incompatibility with one or more other actors, the conflict being in the realm of thought and perceptions, in the realm of emotions or in the realm of will, in a way so that the realization (of one's thoughts, emotions, will) is obstacled by another actor. (Glasl 1977,p.14)

Conflict are thus posed as problems, insatisfactions to which we try to give an answer. The causes of conflicts are many and complex, and much of the possibility of managing them depends on our capacity of analysis and action. Often we quarrel, but apparently for futile things, and sometimes we fall in a vortex that makes things worse and worse and makes us feel more and more bad.

In conflicts, it is very important to have the capactiy of living in difference and sometimes in suffering. In order to dismantle some consolidated behaviours and some wrong habits, one must try to to understand the dynamics and the reality in which they live.

It can be useful to give oneself instruments to analyze the conflicting situations and a common lexicon.

There are four kinds of conflictual action:

1. People who want to pursue different goals. If they are independent persons, this is not a problem: it becomes a problem when these people are instead bound together by some reason. This can happen in all the situations in which a collective or coordinated action is required. It is called DIVERGENCY. For example, a husband and wife might go in vacation together, but they would prefer different destinations.

2. In situations in which many actors concur on the exploitation of a limited resource. In this case, the conflict is defined CONCURRENCY. For example, when many shepherds exploit the same free grazing area.

3. When an actor directs his/her action against the action of another, this is called CREATING OBSTACLES and it's intended to hinder the other in reaching his/her goal.

4. When the action is directed against another agent and not against the other agent's actions, this is called AGRESSION

Competition

Competition (concurrent + obstacling): in real life, it often happens that two agents that concur towards their goals also make obstacling and aggression acts in order to ensure their success.

In a contest, two candidates can concur without competing: for example, in a football match the players cannot directly act on other players: if they make an act of aggression, then it's a penalty.

It is not always easy to distinguish between aggression, obstacling and competition. These cathegories mix in a complex reality. These types are not separated in a clear-cut way: they are a like focal point on a continuous line that starts from a situation of orientation towards an external goal, keeps going towards an augmentation of the intervetions on other people's action, until arriving there where the original goal ends up having a secondary role in front of the will to act on and against the other actor, that is, there where aggression becomes the goal itself. This gradual process is the process of *escalation*.

There are three big phases of conflict evolution: (Glasl 1997)

- *win/win*: mainly cooperative aspects, prevailing over the objective contradictions
- *win/lose*: there is the belief that the conflict can be resolved only in favor of one side; attitudes and perceptions acquire an outstanding importance
- *lose/lose*: damaging the other even at the price of suffering: violence enters the game

This is not a forced path, and our hope is to become able to activate a process of *de-escalation*.

Cooperation

Cooperation means putting on a common table part of one's resources and interests in order to have a collective advantage (which is positive also for the single).

Harmony is realized more easily when interests and goals integrate.

- Altruism is different from cooperation
- Egoism does not correspond to individualism
- You can be egoist and cooperative

Conflicts are usually characterized by the tie between cooperative and competitive processes.

Cooperation is a very interesting element: it offers creative solutions to problems, generating a new "wealth" which is sometimes unexpected. If cooperation is spontaneously perceived, it is evern more insteresting as it can join the elements of freedom and self-realization into a short-term logic, and sometimes even long-term logic.

Threats and explicit promises.

In the case of a normal conflict, the negotiation is articulated with acts of coercition and acts of concession, like threats, warnings and promises. When negotiating, it is wiser to focus on the interests rather than on positions. Because of this, various words have lost meaning.

An example is the threat: a threat has success when the actor that threatens can avoid putting it in practice, because doing so means doing something unpleasant also for oneself, and not only for the other. An example is a strike: when enacting a strike, both parties experience a loss.

Using these instruments knowledgeably, or interpreting them in a correct way when one is subject to them, gives back value and effectiveness to negotiation. Some text define the cooperative game as that game in which the players are able to make binding promises (and then fully exploit the negotiation).

Trust.

Trust is a certain degree of confidence in one's forecasting of the behaviour of another actor, or of the external world.

Trust has an important role in cooperation. It can often substitute the effect of rules and punishments. It happens in a way which is more elastic and effective. The conditions with which we can talk of trust are:

- Non controllability
- Absence of cohercition
- Freedom

Trust is characterized by some factor of risk.

Repeated events.

In real life, social interactions happen often and often with the same people. The best strategies for working together are characterized by these behaviours:

- *correctness*: the first step is cooperative, and one pulls back from cooperation only in response to a pull back
- *forgiveness*: the punitive action is not continuing if the other party starts cooperating again

Non-cooperative strategies ruin the environment that they exploit, draining it of its resources and not permitting actors who use other strategies, as a conseguence also eventually themselves, to survive.

Strategies that gain when cooperating favour the creation of an environment which is favorable and stable over time.

Links.

Conflict resolution Conflict Resolution Network[23] has a number of useful "Free training material".

From "Beyond Intractability: A Free Knowledge Base on More Constructive Approaches to Destructive Conflict":

- De-escalation Stage[24]

[23] `http://www.crnhq.org`

[24] `http://www.beyondintractability.org/essay/de-escalation_stage/`

- Limiting Escalation / De-escalation[25]

From "Intergroup Relations Center - Classroom resources":

- Conflict de-escalation[26]
- Dialogue guidelines[27]

Further reading

This is a list of other useful resources available on the network:

- Informations and code of conduct for Debian mailing lists [28]
- Eric Raymond's "How To Ask Questions The Smart Way" [29]
- How do I quote correctly in Usenet? [30]
- Telsa Gwynne's resources on how to report bugs [31]
- How to report bugs effectively [32]
- How to get useful help on IRC [33]
- Joey Hess "Thread Patterns" tips on how to make sense of high traffic mailing lists [34]

Feedback

If you think that some sections are missing or lacking, please send me a suggestion[35].

It is possible to download a tarball[36] with the original DocBook XML version of the document, which contains many comments with editorial notes and raw material. Also available is a list of recent changes[37].

If you want to work on this document, you can also create a Bazaar-NG[38] branch and we can enjoy merging from each other:

```
$ mkdir dcg; cd dcg
$ bzr init
$ bzr pull http://people.debian.org/~enrico/dcg
```

The Debian Community Guidelines is maintained by Enrico Zini[39]

[25] http://www.beyondintractability.org/essay/limiting_escalation/
[26] http://www.asu.edu/provost/intergroup/resources/classconflict.html
[27] http://www.asu.edu/provost/intergroup/resources/classguidelines.html
[28] http://www.debian.org/MailingLists/
[29] http://www.catb.org/~esr/faqs/smart-questions.html
[30] http://learn.to/quote
[31] http://zenii.linux.org.uk/~telsa/Bugs/bug-talk-notes.html
[32] http://www.chiark.greenend.org.uk/~sgtatham/bugs.html
[33] http://workaround.org/moin/GettingHelpOnIrc
[34] http://www.kitenet.net/~joey/blog/entry/thread_patterns.html
[35] mailto:enrico@debian.org
[36] http://people.debian.org/~enrico/dcg/dcg.tar.gz
[37] http://people.debian.org/~enrico/dcg/changelog.txt
[38] http://www.bazaar-ng.org
[39] http://www.enricozini.org

3. Packaging shared Libraries

Josselin Mouette — CS Systmes d'information

Introducing shared libraries

Basic concepts

A library is a piece of code that can be used in several binaries, split out for factorization reasons. In the old days, libraries were all *statically linked*; that is, they were included directly in the resulting binary. Modern operating systems use shared libraries, which are compiled in separate files and loaded together with the binary at startup time. Shared libraries are the most widespread use of *shared objects*, files containing code that can be loaded at runtime, generally with the `.so` extension.

A bit of terminology

API The *Application Programming Interface* of a library describes how it can be used by the programmer. It generally consists in a list of structures and functions and their associated behavior. Changing the behavior of a function or the type of arguments it requires *breaks* the API: programs that used to compile with an older version of the library will stop building.

ABI The *Application Binary Interface* defines the low-level interface between a shared library and the binary using it. It is specific to the architecture and the operating system, and consists in a lists of *symbols* and their associated type and behavior. A binary linked to a shared library will be able to run with another library, or another version of that library, provided that it implements the same ABI. Adding elements to a structure or turning a function into a macro *breaks* the ABI: binaries that used to run with an older version of the library will stop loading. Most of the time, breaking the API also breaks the ABI.

SONAME The "SONAME" is the canonical name of a shared library, defining an ABI for a given operating system and architecture. It is defined when building the library. The convention for SONAMEs is to use `libfoo.so.N` and to increment N whenever the ABI is changed. This way, ABI–incompatible versions of the library and binaries using them can coexist on the same system.

Linking and using libraries

A simple example of building a library using gcc :

```
gcc -fPIC -c -o foo-init.o foo-init.c
[ ...]
gcc -shared -Wl,-soname,libfoo.so.3 -o libfoo.so.3 foo-init.o \
  foo-client.o [...]
ln -s libfoo.so.3 libfoo.so
```

As the command line shows, the SONAME is defined at that time. The symbolic link is needed for compilation of programs using the library. Supposing it has been installed in a standard location, you can link a binary — which can be another shared library — using it with `-lfoo`. The linker looks for `libfoo.so`, and stores the SONAME found (`libfoo.so.3`) in the binary's ELF[1] header.

The output of the `objdump -p` command shows the headers of an ELF object. For the library, the output contains:

```
  SONAME       libfoo.so.3
```

For the binary, it contains:

```
  NEEDED       libfoo.so.3
```

[1] *Executable and Linking Format*: the binary format for binaries and shared objects on most UNIX systems.

The symbols provided by the library remain undefined in the binary at that time. In the dynamic symbol table showed by `objdump -T`, the library contains the symbol:

```
0807c8e0 g    DF .text  0000007d  Base        foo_init
```

while in the binary it remains undefined:

```
00000000      DF *UND*  0000001c              foo_init
```

When the binary is started, the GNU *dynamic linker*[2] looks for the NEEDED sections and loads the libraries listed there, using the SONAME as a file name. It then maps the undefined symbols to the ones found in the libraries.

Libtool

Libtool is a tool designed to simplify the build process of libraries. It is full of features that make the developers' life easier, and full of bugs that bring added complexity for system administrators and especially distribution maintainers. Its paradigm is to build an extra file, named `libfoo.la`, which contains some metadata about the library; most importantly, the list of library dependencies for the library itself. Together with this file, it can build the shared version `libfoo.so` and the static version `libfoo.a` of the library.

It integrates easily with autoconf and automake. You can put in the `configure.ac`[3]:

```
AM_PROG_LIBTOOL
VERSION_INFO=3:1:0
AC_SUBST(VERSION_INFO)
```

and in the `Makefile.am`:

```
libfoo_la_SOURCES = foo-init.c foo-client.c foo.h [...]
libfoo_la_LDFLAGS = -version-info @VERSION_INFO@
libfoo_HEADERS = foo.h
```

Pkgconfig

Pkgconfig is a tool to replace the variety of `libfoo-config` scripts in a standard way that integrates with autoconf. Here is a sample file, `libnautilus-burn.pc`:

```
prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include/libnautilus-burn

Name: libnautilus-burn
Description: Nautilus Burn Library
Version: 2.12.3
Requires: glib-2.0 gtk+-2.0
Libs: -L${libdir} -lnautilus-burn
Cflags: -I${includedir}
```

The `Cflags:` and `Libs:` fields provide the list of `CFLAGS` and `LDFLAGS` to use for linking with that library. The `Requires:` field provides some dependencies that a binary using that library should also link with. In this case, pkgconfig will also look for `glib-2.0.pc` and `gtk+-2.0.pc`.

Integration with autoconf is provided. Here is an example `configure.ac` test for a package requiring the GTK+ library:

```
PKG_CHECK_MODULES(GTK, gtk+-2.0 >= 2.6.0,,
                  AC_MSG_ERROR([GTK+-2.0 is required]))
```

[2] Other linkers can use a different scheme, especially when it comes to filename lookup.

[3] The version information is given for libtool's versioning scheme. You can read more about it in the libtool manual.

Debian packaging of a shared library

Simple case – what the policy mandates

Packaging a simple library for Debian is not much different from another piece of software. In all cases there should at least be two packages:

- `libfoo3`, containing the `/usr/lib/*.so.*` files, so that you get `libfoo.so.3`. The `postinst` script of this package should contain a call to the `ldconfig` command, and it has to be registered in dpkg's *shlibs* database. This can be achieved by a call to `dh_makeshlibs`.

- `libfoo-dev` or `libfoo3-dev`, containing the headers in `/usr/include`, and other files in `/usr/lib`: the `libfoo.so` symbolic link, the `libfoo.a` static library, and if relevant `libfoo.la` (in `/usr/lib`) and `libfoo.pc` (in `/usr/share/pkgconfig`[4]). It should depend on `libfoo3 (= $Source-Version)`.

The *shlibs* system provides a mapping of library SONAMES to package names and minimal versions for the ABIs a of libraries a package is built against.

Updating the package

As for anything providing an interface, shared libraries have to be treated carefully when it comes to updating the package.

- If the ABI has not changed at all, no changes are required to the package.

- The most common case is the ABI being changed in a backwards-compatible way, by adding symbols. In this case, the *shlibs* system should be informed of the minimum version required. This is achieved by changing the `rules` file to call:

  ```
  dh_makeshlibs -V'libfoo3 (>= 3.1.0)'
  ```

  The referenced version is the one of the latest version where the ABI was changed.

- When some symbols are removed or their meaning is changed, the ABI is broken and the SONAME should have changed. The shared library package name has to be changed to reflect this new SONAME: `libfoo3` becomes `libfoo4`.

- If the API changes, some packages using the library may stop building. If the change is small, it may only require fixing of a handful of packages. If it's a broad change, the simplest course of action is to change the development package name: `libfoo3-dev` becomes `libfoo4-dev`.

Library transitions

Whenever the ABI is broken, a library transition starts. Before anything like this happens, the release team should be asked for approval, so that they know the transition will happen. If possible, two transition implicating the same packages should be avoided, as they would have to complete together.

All packages using the library have to be rebuilt in the *unstable* distribution so that they can go to *testing* together. Depending on the library, the optimal course of action may vary.

- If there is a small enough number of reverse dependencies, things can go fast: an upload right to *unstable*, asking the release team to trigger a set of binary NMUs for all depending packages.

- More complex cases, especially if some reverse dependencies can fail to build, should be started in *experimental*.

- For some nightmare libraries, several source versions are present at once, even in stable releases. The examples of gnutls and libpng come to mind.

[4] Pkgconfig has started moving its `.pc` files from `/usr/lib/pkgconfig` and this should be encouraged.

Providing a debugging version

If the library is known to cause crashes or is under development, the availability of debugging symbols is quite helpful. Fortunately, debhelper can do all of this automatically. After defining an empty `libfoo3-dbg` package, the magic command is:

```
dh_strip --dbg-package=libfoo3-dbg
```

This will move debugging symbols in `/usr/lib/debug` in this package; debuggers like `gdb` can use them automatically.

More complex cases – how to avoid circular dependencies

With large and complex libraries, other kinds of issues appear. Considering the example of `gconf2`, the upstream distribution contains:

- a library used by applications,
- a per-user daemon,
- chunks of data, mostly localization files,
- configuration files,
- documentation,
- support binaries using the library.

To avoid having in `libgconf2-4` any files outside versioned directories, the configuration and data were moved to a `gconf2-common` package. Documentation was put in `libgconf2-dev`, where it is useful, and as mandated by policy, support binaries were put in a separate package, named `gconf2`.

The tricky part is the daemon. When it is not running for the user, it is started by the application using the GConf library, which means the library should depend on the daemon. Still, the daemon is linked with the library. Until 2005, the daemon was in the `gconf2` package, meaning a *circular dependency* between `gconf2` and `libgconf2-4`.

Circular dependencies lead to various issues:

- APT randomly fails to upgrade such packages in large-scale upgrades;
- the `postinst` scripts are executed in a random order;
- worst of all, the `prerm` scripts of depending packages can be executed while dependent packages have been removed. This issue turned out to be a release-critical bug for `gconf2`, seriously breaking the build daemons' environment.

The solution to circular dependencies is to put files depending on each other in a single package: if they cannot live without each other, there is no reason to put them in separate packages. Thus, the daemon was put in the `libgconf2-4` package. To avoid including non-versioned files in the library package, which can be an issue in case of a SONAME change and which will become an issue for the multiarch project, the packaging was modified to use `/usr/lib/libgconf2-4` as its *libexecdir*, putting the daemon in this directory.

Despite having been tested in *experimental*, no less than 6 new RC bugs were reported against the new package. If anything, it means such changes have to be done with extreme care, thinking of all upgrade scenarios; *unstable* users can imagine unsought ways to torture APT and will install any package combination that is allowed.

Common developer mistakes

A commonly spread game among upstream library developers is to keep Debian developers busy. Here are some common ways for them to achieve this goal.

Non-PIC code

As a shared library can be loaded at any position in the address space, its compiled code cannot contain anything that depends on that absolute position. The compiler has to be instructed to build *Position Independent Code* with the `-fPIC` option. Usually, this means building two versions of each code object, one with `-fPIC` and one without. Libtool will do this automatically.

However, some developers using their own build system will forget this flag. Most of the time, they only work with the `i386`, on which non-PIC shared libraries still work. Furthermore, PIC code is slower on this architecture, as it is missing a relative jump instruction, getting some performance fanatics to knowingly remove it.

Non-PIC code can also arise from inline assembly code, if it was not written with position independence in mind. In all cases, lintian will emit an error when finding non-PIC code, which shows up as a `TEXTREL` section in the output of `objdump -p`.

Unstable ABI without SONAME changes

Sometimes, an ABI change is noticed in a released library without a SONAME change. Removal or change of generally unused symbols is the most common case. In such cases, upstream developers will generally not change the SONAME of the library and distributors have to deal with it. The solution is to change the package name, `libfoo3` becoming `libfoo3a`. The new package has to conflict with the old one and all depending packages have to be rebuilt.

Some upstream library developers go even further, not having a clue about what is an ABI. They consider the shared library just like the static version and the ABI can change at each release. Examples include `hdf5` or the Mozilla suite. In case of such an unstable ABI, a simple course of action is to ship only a static version of the library. However, it makes the security team's work a nightmare, as every package using the library has to be rebuilt after a security update.

A more clever solution to such breakage is to give a Debian-specific SONAME to the library and to change it whenever needed. This work has been done for the Mozilla suite in the `xulrunner` package. When the breakage is systematic as in `hdf5`, the change can be automated with libtool, as shows this sample from the diff file:

```
-LT_LINK_LIB=$(LT) --mode=link $(CC) -rpath $(libdir) $(DYNAMIC_DIRS)
+LT_LINK_LIB=$(LT) --mode=link $(CC) -rpath $(libdir) -release $(H5_VERSION) \
  -version-info 0
```

The `-release` flag for libtool gives a string to add to the library name. Thus, the `libhdf5.so.0` library becomes `libhdf5-1.6.5.so.0`.

As for the build process, the library package name has to be changed for each new upstream version: here it becomes `libhdf5-1.6.5-0`. Automated `debian/control` generation helps making updates as easy as with other packages — apart from the fact they have to go through the *NEW* queue at every upstream release.

It should be noted that a clever library design can eliminate most causes for an ABI breakage. An example of such a design can be find in GNOME libraries: all data structures are hidden in private structures that cannot be found in public headers, and they are only accessible through helper functions that always answer to a functional need. Most GNOME libraries haven't changed their SONAMEs for several years despite major architectural changes.

Exporting private symbols

At link time, all functions and global variables that were not declared as `static` in the source code become exported symbols in the generated library. That includes functions that do not appear in public header files, and which as such should not be used as part of the API.

Some application developers make use of this small hole. They define the prototype of these private functions in their own headers and make use of them at link time. Such an application is heavily buggy, as it will break when the library developers decide to change their private functions. To detect these applications reliably and to prevent them from running at all, the list of exported symbols should be restricted. It also helps avoiding symbol name conflicts between libraries.

It can be achieved using a simple version script (see p. 21). There is also a feature from libtool which allows to automate this process. Here is a sample taken from the SDL_mixer Makefile.am file:

```
libSDL_mixer_la_LDFLAGS =       \
[...]
        -export-symbols-regex Mix_.*
```

This way, only symbols being part of the SDL_mixer namespace, those beginning with `Mix_`, are exported.

Namespace conflicts can also occur between symbols from the library itself and functions belonging to a program linking to it. The ELF architecture allows a program to override function definitions from a shared library. The symbols can be protected against this kind of override by using the `-Wl,-Bsymbolic` argument at link time. It should be used for libraries exporting too generic functions, and it should be systematically applied to library plugins, *e.g.* GTK+ input methods or theme engines. Such plugins can have their code intermixed with any kind of application that has not been tested with them, and namespace conflicts should be avoided in this case.

Going further – reducing the release team's hair loss

Versioning the symbols

The problem

Let's consider the following simple scenario: a picture viewer written using GTK+. The software makes use of libgtk for its graphical interface, and of libpng to load PNG images. However, libgtk by itself already depends on libpng. When the ABI of libpng changed, and `libpng.so.2` became `libpng.so.3`, both GTK+ and the application had to be rebuilt. In this kind of case, if only the picture viewer is rebuilt, it will end up depending indirectly on both `libpng.so.2` and `libpng.so.3`.

Here, the software is faced with a design flaw in the dynamic linker: when resolving library dependencies, all symbols found in all dependencies, direct or indirect, are loaded in a global symbol table. Once this is done, there is no way to tell between a symbol that comes from `libpng.so.2` and one with the same name coming from `libpng.so.3`. This way, GTK+ can call some functions that belong to `libpng.so.3` while using the ABI from `libpng.so.2`, causing crashes.

The solution

Such issues can be solved by introducing *versioned symbols* in the libraries. Another option has to be passed at link time:

```
libpng12_la_LDFLAGS += -Wl,--version-script=libpng.vers
```

The *version script* referenced here can be a simple script to give the same version to all symbols:

```
PNG12_0 {
*; };
```

The 1.2.x version (`libpng.so.3`) is given the `PNG12_0` version, while the 1.0.x version is given `PNG10_0`. Let's have a look at the symbols in the libraries using the `objdump -T` command. For the 1.0.x version we have:

```
00006260 g    DF .text  00000011  PNG10_0     png_init_io
```

and for the 1.2.x version:

```
000067a0 g    DF .text  00000011  PNG12_0     png_init_io
```

Now, when a binary is linked against this new version, it still marks the symbols from libpng as undefined, but with a symbol version:

```
00000000      DF *UND*  00000011  PNG12_0     png_init_io
```

When two symbols with the same name are available in the global symbol time, the dynamic linker will know which one to use.

Caveats

To benefit from versioned symbols, all packages using the library have to be rebuilt. Once this is done, it is possible to migrate from a library version to another providing the same symbols, transparently. For a library as widely used as libpng, this was a very slow transition mechanism. Before the *sarge* release, all packages using libpng have been rebuilt using these versioned symbols, whether using version 1.0.x or 1.2.x. After the release, the 1.0.x version has been entirely removed, and packages using 1.0.x have migrated to 1.2.x without major issues. Having waited for a stable release allows to be sure upgrades across stable releases go smoothly.

It is of critical importance to forward such changes to upstream developers and to make sure they are adopted widely. Otherwise, if upstream developers or another distributor chooses to introduce a *different* version for these symbols, the two versions of the library become incompatible. A recent example is found with `libmysqlclient`: the patch was accepted by upstream developers, but they choose to change the symbols version, without knowing it would render the binary library incompatible with the one Debian had been shipping.

Improving the version script

In the case of libpng, it is also beneficial to restrict the list of exported symbols. All of this can be done in a single version script which is automatically generated from the headers:

```
PNG12_0 { global:
png_init_io;
png_read_image;
[...]
local: *; };
```

Restricting the list of dependencies

Relibtoolizing packages

As explained p. 17, libtool stores the list of dependencies of a library in the `libfoo.la` file. While they are only useful for static linking (as the `libfoo.a` file does not store its dependencies), it also uses them for dynamic linking. When the dependencies are also using libtool, it will recurse through `.la` files looking for all dependencies.

As a result, binaries end up being direct linked with many libraries they do not actually require. While this is harmless on a stable platform, it can cause major issues with a system continuously under development like Debian, as dependencies are continuously evolving, being added, removed or migrated to new versions. These unneeded dependencies result in unneeded rebuilds during library transitions and added complexity for migration to the *testing* distribution.

The Debian `libtool` package contains a patch that corrects this behavior. However, as libtool only produces scripts that get included with the upstream package, the package acted upon has to include as a patch the result of a *relibtoolization* using the Debian version of libtool:

```
libtoolize --force --copy ; aclocal ; automake --force-missing --add-missing \
  --foreign --copy ;
autoconf ; rm -rf autom4te.cache
```

It has the drawback to add some continuous burden on the Debian maintainer, as it needs to be done for each new upstream release. Furthermore, it is generally not enough, as indirect dependencies can be added by other sources in a complex build process.

When recursing through dependencies, libtool also adds them to the list of dependencies of the library it's building. For example, when building `libfoo` which requires `libbar` which it turn depends on `libbaz`, it will add a reference to `libbaz` in `libfoo.la`. If the dependency on `libbaz` is removed, packages depending on `libfoo` will fail to build, as they will look for a library that does not exist anymore.

Pkgconfig

Another widespread source of indirect dependencies is pkgconfig. As it also handles dependencies through `Requires:` fields, it will link the binary with several indirect dependencies. Furthermore, developers often add some indirect dependencies in `Libs:` fields.

Recent changes in pkgconfig allow the use of `Requires.private:` and `Libs.private` fields. These libraries and dependencies will be linked in only when using static linking. Here is an example in `cairo.pc`:

```
Requires.private: freetype2 >= 8.0.2 fontconfig xrender libpng12
```

Unlike the relibtoolization, these changes have to be made in the packages that are depended upon, not in the package that hits the problem. Furthermore, it has been argued that libraries that have their headers automatically included (like glib when using GTK+) should be linked in by default nevertheless.

GNU linker magic

The GNU linker has an option that can make all indirect dependencies go away: `--as-needed`. For example, it can be passed to the configure script:

```
LDFLAGS="-Wl,--as-needed" ./configure --prefix=/usr [...]
```

When passed this option, the dynamic linker does not necessarily make the binary it is linking depend on the shared libraries passed with `-lfoo` arguments. First, it checks that the binary is actually using some symbols in the library,

skipping the library if not needed. This mechanism dramatically reduces the list of unneeded dependencies, including the ones upstream developers could have explicitly added.

This option should not be used blindly. In some specific cases, the library should be linked in even when none of its symbols are used. Support for it is still young, and it should not be considered 100 % reliable. Furthermore, it does not solve the issue of libtool recursing in `.la` files and searching for removed libraries.

To make things worse, a recent change in libtool introduced argument reordering at link time, which turns the `--as-needed` option into a dummy one. This only happens when building libraries, not applications. A workaround was developed, as a patch for `ltmain.sh`, for the `libgnome` package where it is of large importance. It is currently waiting for a cleanup before being submitted as another Debian-specific libtool change[5].

Conclusion

Apart from treating each update with care, there is no general rule for packaging shared libraries. There are many solutions and workarounds for known problems, but each of them adds complexity to the packaging and should be considered on a case-by-case basis. As the long list of problems shows, being release manager is not an easy task, and library package maintainers should do their best to keep the release team's task feasible.

There is a huge number of software libraries distributed in the wild, and almost two thousand of them are shipped in the Debian distribution. Among all developers of these libraries, many of them are not aware of shared libraries specific issues. The Debian maintainer's job is more than working around these issues: it is to help upstream understand them and fix their packages. As such, forwarding and explaining patches is a crucial task.

[5]The upstream libtool developers have stated it may be fixed in the future, but not even in libtool 2.0.

4. Codes of Value: Hacker Pragmatics, Poetics, and Selfhood

by Gabriella Coleman, Postdoctoral Fellow, Center for Cultural Analysis, Rutgers University

A note from the editor:
A longer activly maintained version of this text is available at `http://papers.ssrn.com/sol3/papers.cfm?abstract_id=805287`

> I have nothing to declare but my genius
> – Oscar Wilde

```
#count the number of stars in the sky
 $cnt = $sky =~ tr/*/*/;
```

Paper Overview

This line[1] of Perl denotes a hacker homage to cleverness as a double entrede of both semantic ingenuity and technical cleverness. To fully appreciate the semantic humor presented here, we must look at the finer points of a particular set of the developer population, the Perl hacker. These hackers have developed a computer scripting language, Perl, in which terse but technically powerful expressions can be formed (in comparison to other programming languages). The Perl community takes special pride in cleverly condensing long segments of code into very short and sometimes ”obfuscated” one-liners. If this above line of code were to be expanded into something more traditional and accessible to Perl novices, it may read something like:

```
  $cnt = 0;
  $i = 0;
  $skylen = length($sky)
  while ($i < $skylen) {
$sky = substr($sky,0, $i) . '*' . substr($sky, $i+1, length ($skylen));
    $i++;
   }
      $cnt = length($sky);
```

We see that this enterprising Perl programmer has taken 6 lines of code and reduced it by taking advantage of certain side effects found in the constructs of the Perl computer language. With this transformation of ”prose” into terse ”poetry,” the developer displays a mastery of the computer language. This mastery is sealed on semantic level by the joke of ”counting the number of stars in the sky” due to the naming of the variable `$sky`, and the word play of the asterisk or star. Since the counting function is directed to literally count any appearance of the asterisk symbol, a star, (this is what the program does) the programmer decided to display his craftiness by choosing the variable name `$sky` and hence the description of the function ”count the number of stars in the sky.”

This snippet of code is a useful object to present here because it is a potent example of hacker value in a dual capacity. Free and opens source (F/OSS) hackers have come to deem accessible, open code such as the example above and the Perl language it is written in, as valuable. With access to code, hackers argue they can learn new skills and improve technology. But in the above minuscule line of code, we can glean another trace of value. Because it is a particularly tricky way to solve a problem, and contains a nugget of non-technical cleverness, this code reveals the value hackers place

[1] Here is a little more information about the code. The ”tr” in this code is a function that translates all occurrences of the search characters listed, with the corresponding replacement character list. In this case, the search list is delimited by the slash character, so the list of what to search for is the asterisk character. The replacement list is the second asterisk character, so overall it is replacing the asterisk with an asterisk. The side-effect of this code is that the 'tr' function returns the number of search and replaces performed, so by replacing all the asterisks in the variable `$sky`, with asterisks, the variable `$cnt` gets assigned the number of search and replaces that happens, resulting in a count of the number of stars in the `$sky`. What follows after the # symbol is a comment, a non-functional operator found in most programs, theoretically supposed to explain what the code does.

on the performance of wit. This tendency to perform wit is all pervasive in the hacker habitat (Fisher 1999; Raymond 1998; Thomas 2002). Blossoming from the prosaic world of hacker technical and social praxis, the clever performance of technology and humor might be termed as the "semantic ground" through which hackers "construct and represent themselves and others" (Comaroff and Comaroff 1991: 21).

Judging from this Perl example alone, it is not surprising that in much of the literature, hackers are treated as quintessentially individualistic (Turkle 1984, 1995; Levy 1984; Sterling 1992; Castells 2001; Borsook 2000; Davis 1998; Nissembaum 2004). "The hacker," Sherry Turkle writes, "is the defender of idiosyncrasy, individuality, genius and the cult of individual" (1984: 229). Hackers do seem perpetually keen on asserting their individuality through acts of ingenuity, and thus this statement is unmistakably correct. However, in most accounts on hackers, the meaning of individualism is treated either as an ideological cloak or uninteresting, and thus is often left underspecified. In this piece, through an ethnographic examination of hacker coding practices, social interaction, and humor, I specify how hackers conceive of and preform a form of liberal individuality that departs from another version of liberal selfhood.

Hacker notions of creativity and individuality, I argue, extend and re-vision an existing cultural trope of individualism that diverges from the predominant reading of the liberal self as that of the consumer or "possessive individual" (Macpherson 1962; cf. Graeber 1997). Their enactment of individualism is a re-creation of the type of liberal person envisioned in the works of John Stuart Mill in his critique of utilitarianism (1857), more recently addressed in the works of other liberal thinkers like John Dewey (1935), and practically articulated in ideals of meritocracy, institutions of education, and free speech jurisprudence. As Wendy Donner explains, the Millian conception of selfhood sits at odds with a Lockean sensibility "wedded to possessive individualism" for Mill formulates "individualism as flowing from the development and use of the higher human powers, which is antagonistic to a desire to control others" (1991: 130). For hackers, selfhood is foremost a form of self-determination that arises out of the ability to think, speak, and create freely and independently. As Chris Kelty (2005) has persuasively argued by drawing on the work of Charles Taylor (2004), hackers and other net advocates have crafted a liberal "social imaginary" in which the right to build technology free from encumbrance is seen "as essential to freedom and public participation as is speaking to a public about such activities (2005:187). And indeed, the commonplace hacker assertion that free software is about "free speech" not "free beer," signals how hackers have reformulated liberal freedom into their own technical vernacular. Over the last decade, by specifically integrating free speech discourse into the sinews of F/OSS moral philosophy, hackers have gone further than simply resonating with the type of liberal theory exemplified by John S. Mill. They have literally remade this liberal legacy as their own.

Clearly there are culturally pervasive ideals and practical articulations of the Millian paradigm from which hackers can easily draw upon. But the more interesting question is: why does this liberal legacy of the free thinking individual capture the hacker cultural imaginary? In this piece I seek to make this question intelligible by portraying how hackers create value and notions of individuality through routine everyday practices as coding, humor, and interactions with other programmers. It is the symbiosis between their praxis and prevalent liberal tropes of individuality and meritocracy that form the groundwork of hacker liberal self-fashioning as I discuss here. Central to their construction of selfhood is a faith in the power of the human imagination that demands of hackers constant acts of self-development through which they cultivate their skills, improve technology, and prove their worth to other hackers. To become a hacker is to embody a form of individualism that shuns what they designate as mediocrity in favor of a virtuous display of wit, technical ability and intelligence. Hackers consistently and constantly enact this in a repertoire of micropractices, including humor, agonistic yet playful taunting, and the clever composition and display of code.

Since the designation of superior code, cleverness, intelligence, or even a good joke can only be affirmed by other hackers, personal technical development requires both a demanding life of continual performativity as well as the affirmative judgment of others who are similarly engaged in this form of technical self-fashioning. This raises a subtle paradox that textures their modes of sociality and interpersonal interactions: hackers are bound together in an elite fraternal order of judgment that requires of them constant performance of a set of character traits that are often directed to confirm their mental and creative independence from each other.

This paradox alone is not necessarily generative of social tensions. This is, after all, how academics organize much of their labor. However, given that so much of hacker production derives from a collective and common enterprise, a fact that hackers themselves more openly acknowledge and theorize in the very ethical philosophy of F/OSS, their affirmation of independence is potentially subverted by the reality of and desire to recognize collective invention. As I discuss below, the use of humor and technical cleverness reveals as well as attenuates the hacker ambivalence between the forms of individualism and collectivism, elitism and populism, exclusivity and inclusivity that have come to mark their lifeworld.

This piece now continues with a brief discussion of Mill's conception of individuality, self-cultivation, and judgment. This will help ground the second part of the article, which takes a close look at hacker pragmatics and poetics. I open with a discussion on hacker pragmatics as this will help clarify how hackers use cleverness to establish definitions of selfhood. Though this is not on humor per se, in this second half, I also heavily draw on examples of everyday hacker humor, treating it as iconic of the wider range of their "signifying practices" (Hedbidge 1979) through which they define, clarify, and realize the cultural meanings of creativity, individuality, and authorship. The final and third section provides

a closer look at the relation between the hacker self and the liberal self, and ends with a discussion on the hacker ideal of authorship and meritocracy that has grown from their commitment to Millian individualism.

Works Cited

Borsook, Paulina 2000 *Cyberselfish: A Critical Romp through the Terribly Libertarian Culture of High Tech.* New York: Public Affair.
Castells, Manuel 2001 *The Internet Galaxy: Reflections on the Internet, Business, and Society.* Cambridge: Oxford University Press.
Comaroff, Jean and John Commaroff 1991 *Of Revelation and Revolution: Christianity, Colonialism, and Consciousness in South Africa.* Volume One. Chicago: University of Chicago Press.
Davis, Erik 1998 *Technosis: myth, magic, and mysticism in the age of information.* New York: Three Rivers Press.
Dewey, John 1935 *Liberalism and Social Action.* New York: G. P. Putnam's Sons.
Graeber, David 1997 *"Manners, Deference, and Private Property" Comparative Studies in Society and History.* 39(4)694-726.
Hebdige, Dick 1997 *"Subculture the Meaning of Style" in The Subcultures Reader.* New York and London: Routledge, [1979]
Himanen, Pekka 2001 *The Hacker Ethic and the Spirit of the Information Age.* New York: Random House.
Kelty, Chris M. 2005 *"Geeks, Social Imaginaries, and Recursive Publics"* Cultural Anthropology. Vol(20)2.
Levy, Steven 1984 *Hackers Heroes of the Computer Revolution.* New York: Delta.
Macpherson, C. B. 1962 *The Political Theory of Possessive Individualism: Hobbes to Locke.* Oxford: Clarendon Press.
Mill, John S 1969 *Autobiography.*
Nissen, Jorgen 2001 *"Hackers: Masters of Modernity and Modern Technology"* in Digital Diversions: Youth Culture in the Age of Multimedia.
Nissenbaum, Helen 2004 *"Hackers and the Contested Ontology of Cyberspace"* New Media and Society (6)2. `http://www.philsalin.com/patents.html`.
Sterling, Bruce 1992 *The Hacker Crackdown: Law and Disorder on the Electronic Frontier.* New York: Bantam.
Thomas, Douglas 2002 *Hacker Culture.* Minneapolis: University of Minnesota Press.
Turkle, Sherry 1984 *The Second Self: Computers and the Human Spirit.* New York: Simon and Schuster.

5. Cheap Thrills – Instant Inspiration for the Masses

by Meike Reichle (meike@alphascorpii.net)

> In words are seen the state of mind and character and disposition of the speaker.
>
> – Plutarch, Greek historian, biographer, and essayist. (∼45 AD to ∼125 AD) [1]

Introduction

FOSS lives because of the people who use it.

Be they beginners or experienced professionals, everyone can offer valuable contributions and insights to FOSS development and improvement. However, most people don't get up one morning and decide to use some particular distribution or piece of software. In most cases they need to be convinced, or at least provided with a small trigger. One possible trigger can be a talk at a conference or meeting. A presentation about a technical or social topic, delivered by a good speaker can be an inspiring experience - or a dreary and almost agonising one.

On the following pages, I will introduce a set of hints and guidelines on how to give a good presentation. From choice of topic and preparations to the actual delivery and its aftermath, there are many things that should be considered. I'll give general suggestions as well as some personal tips and tricks you may not have come across yet. Everything presented in this paper is either from my own experience as a speaker or talks I've seen. This is a very practical topic and thus this paper does not include everything from the talk which will also feature some practical demonstrations.

Preparation

The weeks before

A good part of a talk's success depends on a careful choice of topic and content. When considering whether to answer a Call for Papers you should ask yourself a couple of questions: Is the topic appropriate for the type of event? Who is the audience? What previous knowledge can you presuppose? What other talks will there be? What is the event's scope? Is it a regular LUG meeting, a conference or a business talk? Once you decided that your talk fits the event and audience, take a minute to also consider whether it fits *you*. Are you sufficiently competent on the topic? From my own experience I can say that "Not yet, but I have n weeks left until the event and will do some reading" is a bad answer to that question. Presenting freshly acquired knowledge is never a good idea. Firstly the reassuring certainty that you have an in-depth knowledge about your topic gives you a calmness that will benefit the quality of your talk, and secondly not being competent at a topic almost always backfires. You take the risk of getting muddled or be confronted with questions you are not able to answer. All of this leaves a bad impression and should be avoided.

The next step after deciding on on a topic and content is preparing the slides for your talk. That is, if you are using slides, since not every talk needs them. If you are giving e.g. an inspirational speech or talking about a social topic you might actually chose to go without any slides. A lot has been said on how to compose good slides, so I'll keep this short and only give a few general guidelines. Firstly, slides are not made so people who didn't see your talk can read up on it later. For that purpose you usually write a paper about your talk, giving the necessary information. I recommend that you always write the paper that goes with a talk in advance. Bringing your thoughts to paper/screen helps to assure they are clear and coherent. If you do not write a paper, write at least a detailed outline before doing your slides. Do not make them up as you go along as it's bad for structure. When designing your slides, keep in mind that you want your audience to do two things at the same time: read your slides and follow your talk. Thus, the two should complete each other. Don't write down every single word you are going to say, too much to read will make people ignore either you or your slides. But also do not limit yourself to a few catchwords. Slides should provide the kind of information that you cannot or do not want to give orally: diagrams, code snippets, URLs or long versions for abbreviations you are using. In order to allow your audience to easily read over your slides as they listen to you, they should be as easily readable as

[1] Hint No. 1: Show them who's boss right at the start. Impress with a brainy quote! Europeans may chose from a broad range of Ancient World writers and philosophers, US citizens may want to go for a dead president.

possible. Chose an unobtrusive design and a large, simple font. Also be careful about using pictures that do not transport any real information (read: funny illustrative cartoons (read: dilbert strips)). They are nice, but do use them sparingly and make sure they really fit. Last but not least: Have as many people as possible proof read your slides. Even the tenth proof reader will still find typos or ambiguities.

Once your slides are done, you'll need to test them. In order to do this you should present your talk at least once in order to check whether it has the right length and a nice flow (no repetitions, sudden changes of topic, lengthy sections etc.). If you want to go for the real thing, present the talk to a bunch of friends or colleagues. This can also be a good way to deal with stage fright. If you cannot find any volunteers, try giving the talk in front of anything else that can serve as a pseudo audience: your hamster (extra points for ignoring the constant chewing, a feature also found at least once among every larger audience), an infant or simply a mirror. Whatever you do, you should at least click through your slides once, mumbling along in some way or another.

This is also helpful when presenting a talk in a language that is not your first language. When writing up slides, you tend to think along in your mother tongue and not notice you are missing a word. You will notice it however when you are actually giving your talk, and so will your audience. So, do look up important/difficult words in advance. If one or a few words are really troubling you, try to find easier synonyms or paraphrase them. It also helps to write a small cheat sheet. (Please *only* use this if you are standing behind some lectern or table, do not fumble them from your pockets when getting stuck.) In any case, if you have trouble remembering something now, you can be sure it will be gone once you are in front of your audience. Finally, a last golden piece of advice: This section is called "The weeks before" for a reason! Please, do yourself and your audience a favour and prepare talks and their accompanying materials well in advance. Do not put them off until the last day. It will be much harder for you and probably also reflect on the piece itself.[2]

The morning before

Once you have made sure your talk will be all nice and shiny turn your attention to yourself. The choice of clothes may not seem important at first, and maybe your audience really doesn't care much. Still, there are a few things to remember. Most of all: Do NOT dress to impress! Dress as you always do or rather, dress like the others. If it is an Open Source event and everyone else is walking around in jeans and t-shirt there's no need to turn up in a suit or costume. If you are talking to business people you'll have to comply with their standards, but still, chose something that comes naturally to you. Forcing yourself into something you feel silly in or that is uncomfortable isn't of much use. The bit more of credibility you gain from your good looks will most likely be outweighed by the effects of the nervousness and discomfort that come with wearing uncomfortable clothing or feeling dressed up. Also, many people tend to get into some kind of mannerism when being concentrated and/or nervous. Thus, if you are prone to fiddling, avoid clothes with strings, buckles or other possible "toys". If you have long hair you may want to tie it back rather than having it in your face. (This also applies if you do not play with it. Your face it what people want to look at, so don't hide it!).

Once at the event (especially if it is a larger event, such as a fair or conference) take some time to wander around and get a feeling for the atmosphere. Check if your talk is still at the time and place you suppose it to be. If possible find the room you'll be talking at and make sure all the equipment is there and working. If you bring your own laptop make sure it works with the beamer, if not, make sure your slides are present and look good on the computer they are presented from. If you are with friends or belong to a booth, do not hang out there until the very last minute. Reserve some time to take a stroll and talk to people. What kind of people are there? How is the vibe? Are people happy with the conference or is there a bad mood? How did the other speakers do? Ask them how their talks went and what the audience is like. Even with everything prepared and a fixed content you can still tweak a talk in order to adapt to the individual situation. Possible customisations include how much humour you use and what kind, which points you stress most or how much time you reserve for questions/discussion. A good speaker will never deliver a talk twice in exactly the same way. Be ready to adapt to the situation and maybe even (if really necessary!) make some last minute changes to your slides.

The hour before

For most people this is the hour of stage fright. Everybody has their own ways to deal with this. My personal advice would be to do something totally different. Play some mindless little game on your laptop, talk to people, take a walk around the block or knit a scarf. Basically, you can do whatever you want but there are a few dos and donts: Don't go for lunch/dinner! A full stomach is no base for a good talk. If you are hungry, have a chocolate bar or something of that kind but not more. If you are prone to stage fright, be assured that whatever you ate, you'll regret it. Also, don't try to drown your nervousness. Be sure you have something (non carbonated!) handy to drink during your talk but be careful before. If you are having a dry mouth beforehand, believe me, drinking won't help. A final hint for the fiddlers: Empty your pockets now!

[2] Believe me, I know. It is currently late at night and well beyond the deadline. I am tired and dozy and can only image how much more eloquent, expedient and enlightening this paper would have been, had I started writing it in time.

Delivery

You have now reached the big moment where you finally step in front of the audience and present your talk. With careful preparation you have already weeded out a lot of possible risks, but of course there are still things to take care of.

Speech

As stated in the beginning of this paper, what you say doesn't only tell people what you want to let them know but much more. Also the way we speak is very much influenced by how we feel. It is thus important to watch the way you speak. My most important advice on this issue is simple: BREATHE!!! Most people tend to get faster and faster when they feel uneasy, simply because the brain says "This is an unpleasant situation and we will not get out of it until we are done with this talk!" so your natural reaction is to speed up. Once you start speeding up, you start breathing wrongly and things go downhill from there. So, breeeaaathe! Slowly. Thoroughly. If you notice that you are getting in a rush, make a break. Take a deep breath and maybe a sip of water. This only takes a second or two, though it feels longer, but greatly helps to calm your nerves and decelerate your talking speed. Also try to listen to yourself. This sounds tricky but the essence is simple: You already know what you are going to say, so the usual behaviour is to always think one step ahead. Try not to do this, it again makes you rush. Instead try to go with your talk as it advances. Study the audience's reactions and try to see whether they can follow you. Actually listening to yourself can also help you spot recurring expressions, something that you are likely to use when under stress. An average person's vocabulary is somewhere between 20,000 and 50,000 words. However, under pressure it tends to shrink to the equivalent of an average marvel comic. So be aware of repetitions and recurring expressions.

Also, don't be afraid of silence! Pauses are mostly a dramatic measure, they are your friends and not a sign of incompetence. They give you and your audience a little rest. You can use them to breathe, gather your thoughts, raise tension, take a sip of water or give people time to read something on your slides. Pauses are totally okay. If you stumble and need to think for a moment, just do so, do not fill every pause or gap with filler expressions or "uuhmm"s and "eeehm"s. Pauses are good.

Body language

Even more telling than your speech is your body language. The first question is where to look. If you are really nervous it is probably better to pick someone in your favour and address your talk at that particular person. In general though I'd recommend not to focus on a single person but let your eyes wander over the audience to get an overall impression. If there is somebody who is really irritating you, try to avoid looking at them. I once gave a talk at which I kept staring at a particulary grumpy looking person for the entire time, thinking about nothing else but how much he seemed to hate my talk. When I was finished, I noticed that I had basically rushed through my talk in half the time I had planned and some people, whose English was not too firm, had not been able to follow me any more since I was talking so fast. So, do not repeat my mistake, if there is a person that distracts you, be it by looking grumpy, fidgeting, talking to someone else or whatever, try to ignore them for the moment. If you want, you can approach them after your talk and often it will show that they actually didn't mean to be as distracting as they were. (In my case it later turned out that the relevant person was totally okay with my talk and simply always looked that grumpy.)

Next is gesturing, how much gesture you use is pretty much up to you. Some people use gestures only sparingly. If you usually are a laid-back person it is absolutely okay to stick to that demeanour in your talks as well. Just take care not to be too boring. Hint: Your pockets are always a bad place for your hands when talking! Some (bad) guidebooks give the advice to practice your talk in front of a mirror, along with the fitting gestures. Don't. It looks plain silly. This is what you do as a major politician or propagandist. With normal people it just doesn't work. Other people (such as me) like to gesture and grimace a lot and basically never stand still. This is okay too, the only problem is that it makes giving talks a bit more exhausting and assures you look funny on pretty much every picture that is taken during your talk.

Same goes for walking around vs. standing still, both is okay, just do whatever fits your usual demeanour. Either way, take care not to block the audience's view at your slides and if you do walk around don't do it too excessively. It makes it harder for your audience to switch between you and your slides.

Managing time

Good timing is an art that is learned by practice and cannot really be caught in simple rules. However, there are a few things you should keep in mind. Firstly, do not overrun your time! It's not fair on the speaker(s) after you and also unpleasant for your audience. They will eventually start shuffling, checking their watches and thus give their uneasy feeling back to you. This is an unpleasant situation you should try to avoid. Check the time occasionally while you talk to see whether you are still in time. This allows you to adapt your speed while talking. If you have a lectern or desk put your watch or an easy to read clock there, so you don't have to interrupt your talk to check the time, but can do it with a

side glance. Also prepare some ”stretchers”, things you can talk about if you have time left and on the other hand think about which bits you can leave out if you get pressed on time. This could be some extra examples, jokes or additional explanations. Try to have several of these small ”crush zones” in your talk so you can adapt its length without having to do obvious things such as skipping slides or adding things at the end.

When planning your talk, you should also consider how much time for questions and discussion it will probably require and how likely it is that you'll have to deal with interjections. On larger conferences and events you are usually given a rough time frame how long you should talk and how much time you should reserve for discussion. As a rule of thumb try to reserve about a third of your overall time for questions/discussions, a bit more if you are giving an introductionary or very controversial talk.

Managing your audience

Your audience is at the same time your best friend and your biggest enemy. First of all though, it's a group of people and thus subjected to group behaviour. Ever noticed that you laugh about much more jokes in a movie when seeing it at the cinema than when seeing it at home in TV? Herd instinct! As a speaker this is one of your biggest allies. If you manage to win the majority the others will eventually follow. These mechanisms can have a lot of effect on your talk and you should keep them in mind. A good example for this are interjections. Dealing with them is not always easy. Generally there is no harm in allowing them, especially if it is a question concerning content. If it is a criticism or some witty comment, things get a bit more complicated. If you are not careful you may find your talk taken over by some jester. Also their example will encourage others and things will eventually run out of hand. So, how do you deal with such interjections? From my experience the best thing is to give a short(!) answer and continue your talk right away. Do not leave a pause for the other to jump in. If the person still keeps disrupting you, point them to the q&a time after your talk. It is always good to encourage comments, but tell your audience that you want them after your talk.

Once your talk is over the time for questions has come. Take care to answer as many questions as possible, do not get caught up in a discussion with a single person or group of people. If such a discussion starts to arise offer to continue it after this session, either in some other location or (if appropriate) on the respective mailing list. If a discussion within the audience comes up do not let it get out of hand, you are still the speaker, you are standing in front and it is your job to manage/moderate such a discussion. Again, if it gets too much, starts hindering other people and their questions or you run out of time, try shifting the discussion to another time and place. As a general rule of thumb you should always provide a possibility for further discussion and comments, e.g. by having your last slide repeat your email address of the address of your or your project's homepage or mailing list. Let people know that you are interested in their opinions and make sure they know where to send them.

The aftermath

Your talk is not over when you leave the room. Apart from eventual continuing discussions the old German football rule applies: ”After the game is before the game.” So, some aftercare is in order. Firstly take some notes on what questions you got after your talk. If there were problems of comprehension try to improve your slides and your talk so these things get clearer. Did people miss things that you forgot to include? Were they able to follow your examples? Did you have redundancies where people got bored? Like a piece of software, a talk is never finished but should be subject to constant improvement. Be approachable after your talk. Many people don't like to state their opinions in public and prefer approaching you privately after the talk. Or maybe you made a really silly mistake and they don't want to publically embarass you about it. So stay around for a bit. Once back at home update your slides and (if existing) your paper. It is best to do this as soon as possible so the memory is still fresh. Also fix spelling mistakes and typos right away so you don't forget them.

Conclusion

We have now reached the end of a very long and maybe even a bit intimidating list of various dos and donts. If you still feel unsure (or didn't but do now) let me assure you that there is no cause for it. Giving talks is, like most things, something that is learned by practice. Rules give you a starting point but eventually you will leave them behind and develop your own tricks and techniques that match your style and taste. Until then I hope the hints provided here will be of some help to you. You do not have to follow them all by the letter. Make choices!

However, if you remember but three things from this talk here is what they should be:

1. Good preparation is key.
2. Breeaathe!
3. NON carbonated!

6. Experiences with CDD's: Centralised Operated Skolelinux Installations at many Schools

by Knut Yrvin – Skolelinux/DebianEdu

The Norwegian Ministry of Education and Research has founded a report about free software in schools. It covers planing and deployment of Skolelinux/DebianEdu that currently includes 234 Norwegian schools, 33,000 client machines, and 101,000 pupils and teachers. The focus is on technical issues, economy and organisation. There is also some feedback on how the teachers use free software in their teaching.

The main conclusion is that no pedagogic, technical, or economical objections using free software in schools remained valid. The numbers show that the municipalities save more when running and maintaining the schools ICT equipment centrally at many schools. The savings are considerably lager with GNU/Linux then with Microsoft Windows. Market prices show that the most cost efficient alternative are Skolelinux diskless workstations. This is at least 40% cheaper than to operate traditional workstation pools (with thick clients).

Experiences from the municipalities show that the cost of running thin clients (ltsp) is more cost efficient than thick clients because it increases the lifespan on the hardware. Surprisingly, the cost of running graphical terminals such as Citrix is the most expensive solution. It is at least three times more expensive than Windows thick clients when comparing prices in the market. When using graphical clients such as Free NX the schools would get approximately the same expenses with operating the system as with Citrix.

The ICT coordinators in the different municipalities say that there are considerable challenges when introducing computer tools in the school. The management at every school must establish a maintenance regime and the teachers must learn to use computer tools actively in the learning process. The activity have nothing to do with the operating system in use. But the schools running free software can afford more equipment, and the lower cost of maintaining the system allows more resources to be used in teachers and the learning process.

Methods

We have used three qualitative methods to get the facts for this report. The first method is called action research. Together with the Municipality in Oslo we have analysed using DebianEdu/Skolelinux as an enterprise solution for all their 175 schools. The second method was to interview the municipalities about their experiences in running Skolelinux and other free software solutions in the schools. The third method was to collect prices in the market both for Microsoft Windows solutions and Skolelinux solutions.

This table shows the municipalities that reported their experiences with centralised managed free software installations at many schools.

Municipalities or City councils	Numbers of locations	Amount of users	Amount of client machines in 2005	Planned client machines in 2008
Hurum	9	1700	200	500
Kongsvinger	9	2300	450	800
Nittedal	10	3200	506	1093
Oslo	175	75,000		25,931
Akershus	31	17,000	6000	6600

Client technology

There are many more client technologies than thin clients, graphical clients, and workstations. There are also diskless workstation, laptops and programs running in a web-browser with Macromedia Flash or Java. And to complicate the matter even more, There are a variety of combinations of the different client technologies.

To get a better picture of the characteristics of client technology, and to describe the pros and cons using the different solutions, we have made a short list. The reason for this is to know if the clients deliver what is expected when used as a tool in the classroom when pupils solve different tasks.

Skolelinux thin clients: Employed with reused computers from 1995 (133MHz CPU) as thin clients with preferably PXE-boot on the network card. No harddisk is required. Requires relatively powerful servers to serves 50 thin clients (dual CPU, 4GB RAM, 100 Mbit/s network) on a switched 100 Mbit/s network.

(+) Give new life to old reused machines. No need to install software on the client. Software is installed centrally on the servers.

(+) Provides a unified infrastructure to users. The clients have reasonable support for graphical applications including sound support. Newer thin client solutions even supports USB memory pen, digital cameras and other devices.

(-) Needs more (powerful) servers than other solutions, but not as much as graphical Citrix or NX terminal solutions

Skolelinux diskless workstations: Use of reused machines from the year 2000 and newer ($>$ 450MHz CPU/256 MB RAM). A harddisk is strongly recommended. Supports DVD, USB memory pen, and other peripherals. A server can support 150 clients on a 100 Mbit/s switched network.

(+) Gives life to newer reused machines, and brand new machines. No need to install software on the clients. Software is installed on one server.

(+) It gives a single architecture solution with the lowest maintenance cost

Skolelinux workstations: Reuse of newer PC's from year 1997 (233MHz CPU/128MB RAM). It requires local harddrive. Needs one server for network services as home directory, print, Internet etc.

(+) Few servers needed

(-) Has to maintain and update software on every client machine

Windows workstations: Reuse of newer PC's from year 2000 (450MHz CPU/256MB RAM). It requires local harddrive. The client machines should be of same type and same production month to be easily managed with software updates and such. The alternative is to use many software images for industrial update of software when security patches and such are rolled out. This gives more man hours adjusting the software for the different client types. Needs one or two server for network services such as file service, print, Internet etc.

(+) Few servers needed

(-) Has to maintain and update software on every client machine

(-) Needs client machines that is the same production month for all the machines to keep the maintenance cost low.

Skolelinux graphical terminals; with NX technology. It is possible to reuse machines from 1995 and newer (133 MHz CPU). The requires relatively more powerful servers, and more bandwidth to the schools. I.e. when the servers are located remotely. The schools also need a local servers for software maintenance on the client machines as well. In practice the graphical terminals has to maintained as workstations with local software. In addition the user application runs on remote servers that gives two structures for software maintenance and extra use of servers and network capacity.

(+) Give new life to older hardware

(-) More servers needed and you'll have to maintain software on every client. That's two solutions for distributing software

(-) Limited support for sound and graphical applications such as video clips and small games etc. Especially when the servers are reside in central remote locations.

Windows graphical terminals: With NX, Citrix or rdesktop technology it is possible to reuse machines from 1995 and newer (133 MHz CPU). This requires relatively powerful servers, and more bandwidth to the schools when the servers are remote. The schools also need a local server for software maintenance on the client machines, that in practice is a workstation with a graphical client. It has almost the same characteristics as graphical terminals with Skolelinux.

(+) Give new life to older hardware

(-) More servers needed and you'll have to maintains software on every client. That's two solutions for distributing software.

(-) Limited support for sound and graphical applications such as video clips and small games etc. This is especially when the servers are placed centrally.

Skolelinux laptops: Use of newer PC's from year 2000 (233MHz CPU/128MB RAM). It requires local harddrive. Needs one server for network services as home directory, print, Internet etc.

(+) Few servers needed

(-) Has to maintain and update software on every client machine. And it has to be a bit more secured when connected to the network.

(-) Laptops are expensive to maintain because of the rough use for the machines that gives short lifespan. gives more work.

This tables give some recommended settings for different types of clients. But first we have to remark on the minimum requirement. It is totally possible to run Microsoft Windows 2000 or Debian Woody on a PC with 64 MB RAM. But it is not usable with an office suite, web browser and a email client. Even with 128 MB RAM on a Windows 2000 PC we have experienced warnings or rejection when printing PDF-formatted documents. ICT operators say the virus scanner uses memory and CPU power. So here is the recommended minimum requirement for hardware that give usable functionality.

System	**Skolelinux thin client**	**Skolelinux PC** [1)]	**Skolelinux diskless workstation**	**Windows 2000 PC**[1]
CPU	133 MHz	233 MHz	450 MHz	450 MHz[2)]
RAM	32 MB	128 MB	128 MB	256 MB
Harddisk	-	1.2 GB	0.5 GB[3)]	2 GB
Introduced	June 1995	May 1997	August 1999	August 1999

1. Could also be used as a client for graphical clients as Citrix, Free NX or rdesktop.
2. Needs more CPU power to handle the virus scanner
3. Small harddisk for local swap is highly recommended

The maintenance and operational cost for a computer network depends on the number of concurrent users and the number of machines and services in use. The routines for software installation and updates have affect on the operation and maintenance cost. The amount of servers also affects the maintenance cost.

Software

Experiences with software usage is that it varies a lot. Some schools have up to 80 different software programs installed on the system. Others have started with 20 and increased it to 50 after different requests from the teachers. Some municipalities have also installed 7-8 Windows applications that are runs with Wine on the Linux thin clients.

The numbers of free software programs for use in schools is increasing. Often it may even seem to be to many programs available. Many ICT-coordinators does an extra job to reduce the amount of programs. It is also an important trend that pupils uses the web browser a lot. The browser is probably the most important tools when using computer programs.

The numbers of user programs that only support one operating system is decreasing. Important examples are FireFox and OpenOffice.org that replace proprietary software. Other programs for use as web applications for use in schools are delivered as web applications for the web browser. The vendors cannot overlook the fact that the use of FireFox is increasing, and they recognise that they must support different platforms to not loose market shares.

When it comes to integration there are two important issues. The first is about using OpenDocument file format. A lot of pupils have access to their parents PC's at home. One or both parents has a PC from their employer with Microsoft Office. The parents wont allow the pupil to install OpenOffice.org on their PC. And that gives some objections concerning use of OpenDocument. There are two strategies to work around this. The one is to set Microsoft Word as the standard format when saving in OpenOffice.org at the schools. The second is to explain to all the parents how they should

handle this issue. Secondary schools save with OpenDocument as the standard because older kids are more experienced to handle different file formats compared to younger pupils.

The other integration issue is combining of different Linux-distributions or integrating with Microsoft Windows. Experiences show that it is really easy to connect serves with K12LTSP onto a Skolelinux network. The same for Windows PC's which can be connected to the Skolelinux servers with SAMBA out-of-the-box. When it comes to the City Council of Akershus they have a Windows network with Active Directory. They connect Skolelinux thin client servers to that net with a tailored SAMBA connector made for all their 31 schools. That way they can reuses a lot of client hardware that would have to be thrown away if they had to run only Windows. Also Mac PC's from Apple are able to connect to the Skolelinux net following a simple howto. There are also solutions to connect Skolelinux thin client server to a net based on SuSE Novell. This table gives an overview of different client solutions that can be easily integrated into a Skolelinux network:

Server solution	**"Client solution"**
Skolelinux	Skolelinux Lessdisk Workstation (LTSP)
Skolelinux	Skolelinux thin clients (LTSP)
Skolelinux	Skolelinux Workstation
Skolelinux	Windows Workstations
Skolelinux	Free NX (Citrix)
Skolelinux	Mac OS X
Skolelinux	K12LTSP (LTSP server)
Windows 2000/2003 Server with Active Directory	Skolelinux thin client server
Skolelinux	Linux laptops (Kubuntu, Skolelinux, etc.)
SuSE Novell with eDirectory	Skolelinux client server

Remember that laptops are expensive to maintain

The last technical issue is that OpenOffice.org, FireFox and some of the windows systems are bloated. This results gives in more expensive memory usage on the servers and the client machines without giving real usability. Some of the municipalities have changed their Windows Manager to ICEwm. Others have replaced OpenOffice.org with Koffice to get an easier to use editor when writing short text documents. The teachers say it is not a big problem.

The pupils don't really need a fully fledged office suite made for office employees when using ICT tools in lower grades. Most of the pupils uses computers for something other than using spreadsheets and presentations. This is the same situations for grown-ups. Most the work done with computer tools at the workplace is done by tailor made professional tools[1] with 42% in the service oriented sector and 89% in the industrial sector. 15% of the time when using the PC is used on a word processor and 2% of the time on a spreadsheet. So the office application are used 17% of the average time on 2 h:15 min spent in front of the computer.

The pupils use computers to make music, sending drawings to school classes abroad, and to learn to write (decode the words). So here lightweight technologies meets usability for pupils in the lower grades.

To sum up the situation with software. Some schools use a lot of free software. But the real killer application is the web browser with different online tools tailor made for teaching. Second comes the office application. That's because a lot of teachers believe the pupils needs to learn an Office suite, but a lot of teachers prefers more lightweight applications. It is issues with handling OpenDocuments at the schools when the pupils parents has a job computer at home with MS Office. This has to be addressed. Experiences from most of the municipalities and city councils is that they integrate different solutions in a cost efficient way.

The first one is that more and more applications are made platform independent by serving them through a web server or that applications is usable with platform independent libraries (Qt, Gtk#, Java).

Investments in equipment

This table shows the real investment in terminal technology and servers in three of the municipalities. The cost of building the network and switches are not included. But the cost for every connection is pretty standard with 255 Euro for every PC, that gives cost at of 54 Euro annually over a 5 year period (with 2% interests).

[1] A Journal du Net: http://solutions.journaldunet.com/0409/040915_etude_postedetravail.shtml[2]

Hardware investments to 2005

Art	Nittedal	Hurum	Kongsvinger
Client machines (amount)	# 506	# 200	# 450
Servers (amount)	# 11	# 10	# 20
Purchase of reused client machines	17,818	8,909	26,727
Purchase of new client machines	18,327	30,545	45,818
Purchase of new servers	75,568	26,600	63,636
Software Licences			
Hardware investments	111,713	66,054	136,181
Network switch, electric switch	128,799	50,909	114,545
Training cost of ICT contacts and operators	7,636	2,545	6,873
Total	**248,148**	**119,508**	**257,598**
Hardware and training for each client	**236 Eur**	**343 Eur**	**318 Eur**
Network switch, electricity each client	**255 Eur**	**255 Eur**	**255 Eur**
Annually cost over 5 year	**104 Eur**	**127 Eur**	**121 Eur**
Euro (100 EURO = 785.72 NOK)	7.86		

The municipality of Nittedal has the lowest investment cost in client hardware compared with the number of clients. The reason for this is that they have reused more equipment than the other municipalities. They do a bigger effort in getting reused PC's from the administration, install network cards with PXE (Preboot eXecution Environment (Intel)). This is done by the local ICT contact at each school. Other municipalities does this work with their central staff at the ICT service at the town hall.

The ICT service in the different municipalities also report the prices of hardware in the marketplace. Graphical terminals tailor made for Windows cost between 381 – 445 Euro (without a monitor). A tailor made Linux terminal cost 234 Euro with a reused monitor. Reused client machines cost between 115 – 210 Euro with monitor. Adding on a new server and the necessary switches you get a package that can be bought with Skolelinux out of the box. Solution prices from the InOut[3], a reseller of reused computers to schools with server(s) and switches:

Skolelinux solution with 100 complete clients	Economical package with 10 complete clients
2 servers (new). Installed and configured	1 server (new). Installed and configured
# 100 client (PC, monitors, keyboard, mouse)	# 10 client (PC, monitors, keyboard, mouse)
Network switches (reused)	Network switches (reused)
Pedagogic software	Pedagogic software
Price for the package (w/o wat) 24,945 Eur	Price for the package (w/o wat) 3,169 Eur
Price for each client (w/o wat) 249 Eur	Price for each client (w/o wat) 317 Eur

In Norway also FAIR[5] (Fair Allocation of Infotech Resources) helping the developing world with supplying reused ICT equipment to schools. Some of that equipment has Skolelinux installed[7]. as this newspaper article shows. They doing quality assurance of the computers in Norway before they are shipped to other countries. The French charity Emmaus also providing reused computers to public and private institutions. They are actively participating in the Skolelinux project from France[9].

Running cost

Running cost depends on various factors. The most noticeable are the amount for concurrent users, and how the software is distributed. Where the servers and services are placed has some influence on the operational cost. And what kind for service level agreement that's required. But first some background to understand the running cost of the ICT services that operates Skolelinux in all or many schools in their

Reactive problem control is to fix the problems when it appears. If the computer is infected by a virus one of the common strategies is to reinstall the system. Does the pupil want a program found on the Internet, it is just to install without asking if the program is relevant to for teaching purposes.

[3] InOut reseller of reused computers: http://www.inout.no/skolelinux/[4]

[5] FAIR (Fair Allocation of Infotech Resoruces) http://www.fairinternational.org/[6]

[7] Skolelinux from Norway to Eritrea in Norway's largest ICT transfer ever: http://europa.eu.int/idabc/en/document/3355/469[8]

[9] http://schlossgul.org/wakka.php?wiki=PagePrincipale[10]

Advantage 1: Could be reasonable running cost if every computer has the same version of the operating system. For Windows it is necessary to have identical machines for efficient handling of image technologies.

Advantage 2: The schools could have stored ready-to-use PC's that could replace faulty machines.

Disadvantage: The consequence of different incidents demands a lot for work or gives long delays before it is fixed. E.g full disks.

Proactive problem control is to detect and fix the errors before the users experience it. One example on proactive problem control is to update disk image every night or weekly. When the pupils log into the system the day after, everything is like the school teachers want's it. The ICT operator get warnings about errors and such, and fix the problems before the users notice that some thing's are wrong. E.g the ICT operator gets a warning that a one of the mirrored disks is corrupted, and needs to be replaced.

Advantage 1: The system gets high uptime and stability given the right tools and skills to run such an operation. It gets more easy to handle more amount of computers

Disadvantage 1: Expects more skills running the solution. Gives more cost when establishing and operating the solution.

Disadvantage 2: Proactive is more expensive than reactive problem control. What to choose depends on the consequences when the system is down. It is difficult to calculate the cost of lost work in the classroom if the tools don't work. Does the users want little downtime, it is necessary to invest in high uptime.

In general reactive problem control is not recommended if the goal is to have satisfied users. The only argument in favour for a reactive operation is (1) price or (2) lack of resources and/or competence. Reactive maintenance is not a thing that is chosen. It is something that is enforced on the schools.

Scalability is important for how to get the most of the investment. When the number of machines increases the cost could rise more than linear. A lot of things has to be taken care of when centralising the ICT operation. One of the most important factors when centralising is network capacity. But also where to place the services.

The Norwegian Educational Communications and Technology Agency gives this advice about placing services.

General recommendations to improve the client operation	**Thin clients Lockdown of the clients Local servers**	**Remote operation Centralisation of certain services**	**Reginal or national servers Centralisation of all maintenance**
The network capacity	Low bandwidth (IDSN)	Medium bandwidth (ADSL etc.)	High bandwidth (Fibre 1 Gbit/s)

The services that can be handled centrally independent on the bandwidth:

- Configuration management. E.g to have control over the configurations of machines, network, programs and services
- Software management. To have an overview and control on the use of programs and if the performance of programs and services
- Updates and security patches
- User administration. In the nordic countries the governments has got together to make a common LDAP schema for pupils. In Norway this is a part of the Federated Electroic Identety management[11]
- Licence administration
- Surveillance and measurements

Service level is also a part of the solution. Does the ICT service provide reactive problem management or proactive? What is expected when things go wrong. The ICT coordinator for the schools in Akershus City Council says that the don't want to pay for the extra cost for fail over. If a disk fails on a server when doing the exam, they would use the day to change the disk, and restore from backup. It is the same they do if one school has water leakage. That has happened.

[11] FEIDE - Federated Electronic Identity: http://www.feide.no/index.en.html[12]

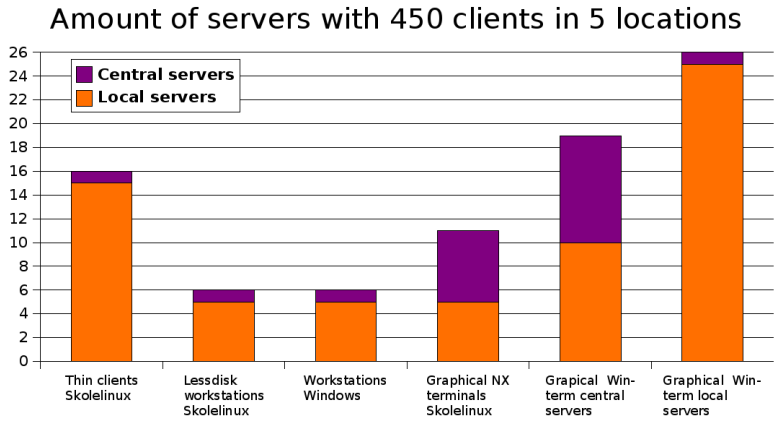

Figure 6.1.: Server Client Ratio

They fix it, and the next day they carry through the exam. The city of Oslo has a service level agreement with guaranteed uptime, and expected recover after machines crash or goes off-line. The guarantees is valid for approved workstation hardware, not for graphical Windows clients (Citrix) on reused computers.

The graph Server Client Ratio gives an overview on the amount of standard servers that are needed for different client solutions. The example is based on 5 schools with 320-450 pupils and 90 clients at each location. The graph is based on real life experiences when running different client solutions at many schools in Oslo, the main city of Norway.

It is mostly technical reasons for the different number of servers with the same amount of clients. In the City of Oslo they mainly have 2-8 Mbit/s broadband to the schools with their graphical terminals on Windows solution. They also have to support workstations to multimedia applications because of the limited performance on graphical clients when using multimedia. Then they has to place out two servers at every school. This is in addition to the centralised placed servers to run the applications that is distributed with graphical terminals to the schools.

With use of graphical terminals the ICT service also get two architectures with running the software, maintenance and updates. The software runs on central servers and the ICT service also has to maintain software on local reused machines and workstations. This gives most of the explanation behind why the operational cost is notably higher with graphical terminals with Citrix or Free NX than other client technologies. That said does some municipalities support Free NX clients to pupils that have Windows or Linux at home as an additional service. It is easy to set up and the ICT staff don't need to maintain the client machines at home. If nothing works at home, they don't need to support it.

Operator cost in 2005

The tables that follow gives an overview of the positions and man-hours used to maintain and run the ICT system at the schools in our survey:

Municipalities City Councils	**ICT operator centrally**	**ICT pedagogic coordinator**	**ICT contact at every school**	**Sum**
Municipality of Hurum (200 clients)	position		8 % position (2:40 h a week)	1,9 positions
Municipality of Kongsvinger (450 clients)	position	position	10 % position (3:20h a week)	1,2 positions
Municipality of Nittedal (506 clients)	position	position	6 % position (1-2h a week. Request for 4h a week)	1,6 positions (request for increasing ti to 2,1 position)
City of Oslo in 2008 (25 931 clients)	External ASP	2 positions	30 % position	*
Akershus City Council (6 600 clients)		80 % position	70-80 % position. Some also have apprentice	*

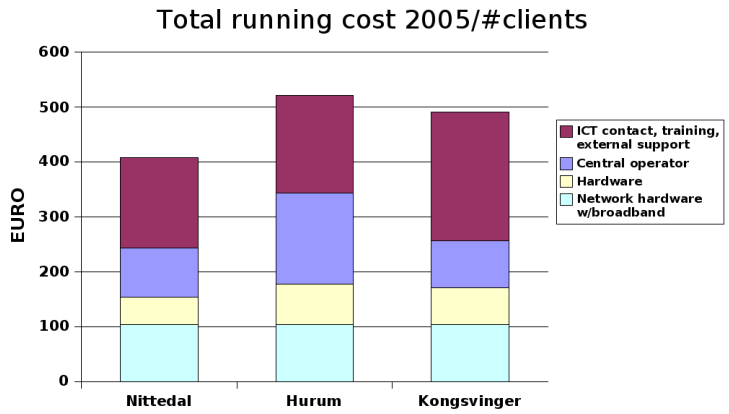

Figure 6.2.: Total running cost 2005

clients = client machines

With help of the three of the municipalities we got the overview of the operator cost in 2005:

Operator cost in 2005 with a Skolelinux network

Art	Nittedal	Hurum	Kongsvinger
Number for clients	# 506	# 200	# 450
Central operator	25,773	25,773	25,773
ICT instructor	25,773		25,773
ICT contacts	29,454	35,345	53,946
External support	19,091	7,636	12,727
Training	28,000		25,454
Sum	128,596	68,954	144,122
Running operational cost/PC	**254 Eur**	**345 Eur**	**320 Eur**
Central operator cost/PC	**89 Eur**	**167 Eur**	**86 Eur**
Euro (100 EURO = 785.72 NOK)	7.86		

It is some important things to say about the persons that runs the ICT operation, and their different roles. The most important is the division of work done helping teachers using the ICT tools in education. That's often a totally different effort than build, operate and maintaining the ICT installations. So most of the municipalities has this arrangement with dividing the pedagogic effort and the technical. Some of the municipalities have a teacher with grate ICT skills with e.g certification in Red Hat and years of experience running Windows. Then they use that person as a project leader building up the operation.

In short the different roles are ***ICT instructor*** or a ***coordinator***. This person works with the pedagogical aspects helping out headmasters and ***ICT contacts*** at the schools motivating teachers to use ICT tools in their teaching. ICT contacts helping out with technical issues as changing defect client machines, replace a broken keyboard or giving feedback about different request for software and services. The ***Central Operator*** is in general a person that works in the ICT service housed a central place in the municipalities.

Total running cost in 2005

This graph shows the total running cost in 2005 for three of the municipalities we have investigated.

The graph clearly shows that the biggest running cost for the work done by people is about 60-70% of the total running cost. The equipment cost annually around 30-40% of the whole operation.

Total operator cost in 2008

The City of Oslo has at least 10 schools that runs Skolelinux today. Some of them got an external service provider to operate the solution with competitive prices. When calculating the future cost of a enterprise Skolelinux solution this was

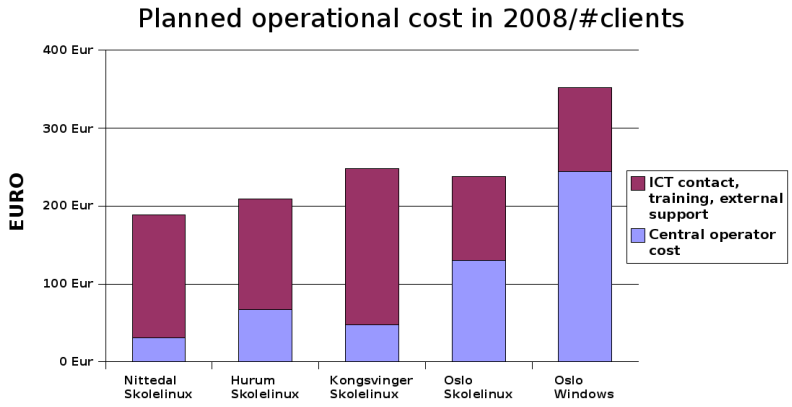

Figure 6.3.: Planned operational cost in 2008

taken into account. We also used the experiences serving 70.000 users accounts with e-mail, backup and single signon on 5 platform at The University of Oslo. Doing that we got a solution that was in accordance with the requirement from the City of Oslo.

The City also had a external service provider with an 5 year agreement to run and maintain the client machines and serves in the Schools. So the prices running todays Windows based system is known from the contracts with the educational department in Oslo. With this in mind we asked what the running operation would be for the other municipalities in 2008. Most of the municipalities had wide expansion plans already decided. The budgets was also in place. So here are the numbers:

Operator cost in 2008

Art	**Nittedal Skolelinux**	**Hurum Skolelinux**	**Kongsvinger Skolelinux**	**City of Oslo Skolelinux**	**City of Oslo Windows**
Number for clients	# 1,093	# 500	# 800	# 25,931	# 25,931
Central operator	25,773	25,773	25,773		
ICT instructor	25,773		25,773	51,545	51,545
ICT contacts	117,817	70,690	107,891	2,705,799	2,705,799
External support	7,636	7,636	12,727	3,326,885	6,283,409
Training	28,000		25,454	79,545	7,545
Sum	206,092	104,599	198,418	6,163,773	9,120,297
Running operational cost/PC	**189 Eur**	**209 Eur**	**248 Eur**	**238 Eur**	**352 Eur**
Central operator cost/PC	**31 Eur**	**67 Eur**	**48 Eur**	**130 Eur**	**244 Eur**
Euro (100 EURO = 785.72 NOK)	7.86				

One important issue is that the enterprise Skolelinux solution designed for Oslo will have requirements for service level that's higher than the other municipalities. It also has different client solution with Skolelinux diskless workstations. The latter is not on to on comparable to the Citrix graphical clients or workstation with Windows most of the schools uses in Oslo today. Most of the other municipalities are heavily users of Skolelinux thin clients with LTSP. The cost of running this in Oslo it is not much higher than diskless workstations, but it is some more expensive.

That said the graphs shows that the operational cost without hardware an network will decrease compared to the situation in 2005. The reason for this is mainly the scalability built in in the Skolelinux architecture. A central ICT operator is able to scape it is effort to handle an doubling of client machines without doubling the amount of work hours.

Market prices

In the summer 2005 we called some companies that provides professional support to schools with maintenance agreements. They act as standard application service provider serving customers running Windows systems or Skolelinux based solutions. We asked for the marked prices and how the prices varied depending on the numbers of supported clients. The reason application providers look to the number of clients is that this produces most of the work when supporting the ICT contact at the school.

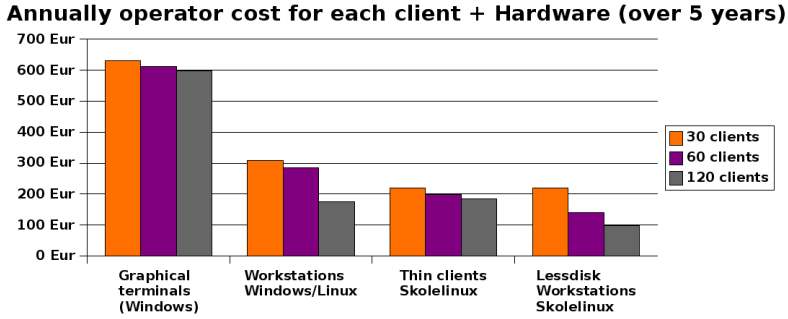

Figure 6.4.: Annually operator cost for each client

Also the number of servers and the places where software is updated determine the prices as we can see from the prices concerning graphical terminals. The cost for the local ICT contact at every school ICT instructor is not calculated in. The cost for the network switches, electricity and broadband is not calculated in either. When doing this limitations the operational cost really shows the difference with different client solutions and some advantages running more clients at the same site.

Conclusions about costs

The calculations and analysis done by the City of Oslo shows that it is cheaper to run a ICT solution in the schools maintained from a central place with local ICT contact helping out with easy tasks. A enterprise solution with Skolelinux is considerable cheaper to operate than a Windows solution.

Experiences with centrally operated Skolelinux solutions in schools is that the operating cost is decreeing a little for every client machine when increasing the amount of PC's. That tells us that the solution is scalable when it comes to operational cost. The task helping users at every school is connected with concurrent users. This job is mainly done by the ICT contact. When increasing the amount of client machines from 50 to 100 the extra workload should not not be increased to more than a day a week when running a Skolelinux solution managed from a central ICT service.

When it comes to marked in the marked it shows that Lessdisk Workstations (lessdisks[13] or LTSP[15]) are 44% cheaper to operate than any other client solution. Windows and Skolelinux workstations comes on a second place when using reused computers and not calculate in the extra expenses with different machines and the software licences using software from Microsoft. To keep the operational cost with Windows low identical machines is recommended, and that's difficult over a period in 1-2 years because the manufacturer changes the hardware specifications all the time.

The thin client alternative with Skolelinux is cheaper than workstation in smaller installation or when reusing hardware as the main strategy at all the schools, as the municipalities shows us. The most costly alternative is graphical terminals with Windows (Citrix).

The risks

Before saying anything about technical risks, I'll focuse on the teachers role and competence.

The biggest risk when introducing software in the classroom is the human factor. An ICT service or a developer seldom know the requirements for use of software in the classroom. Technicians are often over-focuses on technical aspects. The teachers or principals are not specially known with use of software as a tool when teaching. Some teachers also use the computers as a reward for pupils that's fast in solving equations on paper in the math class.

Others don't really know where to start and they end up using computers to present traditional learning material with presentations made in OpenOffice.org or MS PowerPoint. Some ICT instructors have expressed they concern by this story:

We have placed out the "car" in the classroom. Now the teachers has to learn to drive it. Or else it is just a expensive toy, that is standing there with the screensaver on.

In Norway it is regulated by law that the employed and the union representative should be kept oriented about the systems that are used to plan and to do the job, also planned changes in such systems. Training should be provided to get to know the systems, and they should participate in developing the solutions.

[13]The Lessdisks project: http://lessdisks.sourceforge.net/[14]
[15]LTSP MueKow project: http://wiki.ltsp.org/twiki/bin/view/Ltsp/MueKow[16]

There are big differences in the results in teaching the teachers even if it is used a considerable amount of resources on this. In the municipality in Nittedal the priorities to the headmaster is important. A centrally placed ICT instructor in half a position has difficulties reaching out to 100 teachers in 10 different schools. Therefore they need external resources. For the teachers to following up the courses, it has to be set aside time to that. The work with learning ICT has to be followed up. Learning in small groups with 3 to 4 teachers has been more successful than individual self study. The headmaster has to motivate the teachers. Nothing of this are dependent on operating system. Skolelinux is not more difficult to learn than Windows the ICT instructors says.

When it comes to the ICT operators few of them was known to Linux or Unix before they started. They taught them selves, used informal networks, and spread the knowledge when rolling out the solution at the schools. The local ICT contacts also lean from the ICT operator when visiting the schools doing maintenance. The cost of this is usually covered by the ordinary measures. Even if there are differences in background and experiences with the ICT operators running Skolelinux centrally, they have all got help on e-mail lists and tips over Internet.

Stability, security and upgrades

The municipalities report 100% uptime between 8 and 16 hours when there are pupils and teaching at the schools. It has been downtime at some of the schools because of some network plugs that was not properly connected to the network card. When it comes to handling viruses some of the municipalities has installed Clamav[17]. It has not been any virus attacks on Skolelinux. But they have identified some Windows worms that was aimed at MS Internet Explorer.

There are some activities to stop unwanted web-sites with the firewall. The are routines where the teachers reports unwanted web-sites, and the ICT operators submit this sites to the firewall.

When it comes to password handling it is important to have easy password for the pupils in the lower classes. Also it is almost an requirement that teachers can change password for pupils that has forgotten it. It is to much work sending a request about password to the central ICT operator to send this back to the teacher that then gives it to the pupil. So there are essential to have a delegated function that teachers could change pupils password without root access.

At last there are not wise to upgrade the installations in the schools semester. The municipalities most of the times does upgrades at the end of the semester, in the schools holidays. It is recommended to use the stable and same version of Skolelinux on all the machines. That's makes less worries.

Integration and user programs

When it comes to integration there are a lot of possibilities that are in use. One of the municipalities uses Skolelinux as the main server and K12LTSP as the LTSP solution. It is connected to the authentication service in Skolelinux LDAP with a standard Red Hat tool. Others use Skolelinux as a thin client server in a Windows 2000/2003 network integrated with Samba. A lot of schools connects Windows Workstations to the Skolelinux network.

A lot of questions are asked about the user programs. The municipalities that uses Skolelinux has no special concerns when it comes to most of the programs used in the classroom. Mostly because of the widely use of the web browser to gather information, or use of platform independent web applications.

But when it comes to OpenOffice.org there are some practical issues trying to open OpenDocument in MS Office at the home computer where one of the parents has got Microsoft programs from their job. They don't want to install OpenOffice.org. The schools and municipalities has 3-4 strategies handling this. Some municipalities just use OpenDocument as default and sending the pupils a CD home with OpenOffice.org.

Other places the teachers explains at the parent meetings that the real problem is that Microsoft does not really interoperate with the world because the pupils don't have problems with MS Office documents at the school. The problem is at home, so they should really install OpenOffice.org. Not all are able to explain this, so some of the schools just set MS Office as the default storage format when using OpenOffice.org at the school. With OpenOffice.org 2.0 that don't produces any problem the municipalities reports.

When it comes to web applications in Norway the most important vendors testing their application against different Linux distributions as Skolelinux and Red Hat. Also The Ministry of Government Administration and Reform requires that every governmental services on the web should be accessible with open standards[19]. This policy has broad political support both from the left, right and the centre in the political landscape[21]. From a commercial point of view there are now risky to not support open standard and say no to potential costumers in public sectors that stands for around 20% of all the acquisitions in Norway.

[17] Virus scanner Clamay: http://sourceforge.net/projects/clamav/[18]

[19] Open Source in the Norwegian Government: http://opensource.phpmagazine.net/2006/04/open_source_in_the_norwegian_g.html[20]

[21] Norwegian Minister: Proprietary Formats No Longer Acceptable in Communication with Government: http://europa.eu.int/idabc/en/document/4403/469 [22]

Translation

Norwegian is a little language in an international perspective with 4,6 million inhabitants and two official languages. In addition Sami is a co-official language of six municipalities and Finnish is used in one municipality. When most of the computer programs are almost instantly translated into the languages who many speaks as German, French or Russian, proprietary vendors don't see the need for translate their applications into the two official languages in Norway.

So what's done voluntarily has an impact. The Norwegian bokmål and nynorsk has good translation both in KDE and GNOME. KDE was translated to nynorsk in 2001. This inspired to translate OpenOffice.org to the two Norwegian languages. Then the Norwegian parliament decided in 2004 that every program used in education should be available in both the official Norwegian languages. Many says that this was possible because of the results done by the voluntaries. Since OpenOffice.org is translated it's used at many schools, also on Windows. And a lot of pupils copy it freely to use it at home.

There are initiatives that have professionalised the translating effort for some applications. e.g there is established a foundation for promoting free software office applications. They handles the translation of OpenOffice.org to Norwegian bokmål and nynorsk. The cost doing this is just a fraction of what the City Councils pays for Microsoft Office in the secondary schools. The average cost to translate and maintain the OpenOffice.org translation for two Norwegian languages cost 254 544 Euro annually. That's just a 4 to 5 times less than MS Office licences cost in most of the secondary schools in Norway, and then we are speaking of the special prices in the education.

Even if most of the free software in use at the schools are translated, some municipalities conclude that it is has some risks to be dependence on the voluntary effort. There has to be done work to maintain the translation in free software and the translation work and quality insurance of the translations should be paid for.

Licence revision

In 2004 the City Council in Akershus paid external consultants 31 818 Euro to count Microsoft licences at 31 schools. They also had the same cost internally to aid the counting done by the external firm. The municipalities that runs Windows tells they use 1-2 days a month to keep track of their licences in their schools, and 3-4 weeks when they do a licence revision. When we also know that a central ICT operator often has half a position to maintain 500 client machines at 9-11 schools, at least 2-3 weeks is used counting licences annually.

This practice has really upset some of the municipalities. The municipalities that are using free software have no extra cost concerning licence revision. As a marketing effort Microsoft says that schools with a MS School Agreement don't need to do licence revision when paying for all the machines, also them with Mac OS, Linux or FreeBSD. The companies school licences has been like this science 1998 a Microsoft spokesperson has told the press.

Paying Microsoft for using Linux, Mac or FreeBSD has upset a lot of people. Members of the Parliament has asked questions about the practice to the The Minister for Competition. The debate about this was high in the Norwegian computer magazines in December and January 2005/2006 after 12 out of 19 City Councils agreed on the Microsoft licence terms for schools without a public tender. The Minister wrote a letter to one municipality. She recommended to report the City Councils to The Norwegian Governmental Office for Complaints on Public Tenders. The Norwegian Competition Authority are for time being investigating Microsoft for their practice. The Competition Authority says that the schools in Create Britain could not sign this kind of agreement.

Maintaining free software

It is a lot of fear, uncertainty and doubt (FUD) spread about Skolelinux. One of the most important is that it is a voluntary community effort that will die when the founders get other things to concern about. Even if the solution is a part of Debian there are a lot of histories that weakens the perception of our effort. It is always questions about updates, maintenance, how to start, and training. E-mail lists with volunteers is just not good enough. The schools will need professional support on daytime. Someone to keep accountable. That's not so much concerning the distribution, but more how to get help if something fails or break.

To ensure that people work to a common goal, that's difficult enough because all the free developers have their own motivation, and often no money, there is established a foundation that owns a professional service provider. They working actively to ensure a sustainable environment in cooperation with different companies and institutions. Also the international relations are of decisive importance as:

- Custom Debian Distributions with DebianEdu/Skolelinux branches of active developer communities in France, Germany and Greece. Also the cooperation with gnuLinEx in Spain (Extremadura), PSL-Brasil in Brasil and tuxLab in South Africa is of importance
- LTSP.org and Lessdisks are projects that support diskless workstations. The work that is done by the developers in (k)ubuntu, ltsp-team and Skolelinux really gives improved and easy to maintain client solution

- edubuntu project is a good thing running ahead improving the desktop experience. They also want to deliver a municipality wide server solution for schools in around 12-18 moths. Skolelinux and Edubuntu developers work together on different project to improve the offer to the schools
- Cooperation with resellers of reused computers. They sell solutions with making it really easy to acquire Skolelinux solutions for a whole school or a municipality

All this is done to fight of the impression that you need to be a Unix operator with sandals and long hair to maintain a large Skolelinux installation. The main purpose is to get it safe to use free software in a teaching environment.

Recommendations

In general free software gives a complete solution to the schools needs both pedagogically and technically. It is of grate importance to further educate the teachers in using to support an active learning process with ICT tools in the different subjects. The principals has to follow up on this because a lot of teachers has forgotten how to learn new things themselves. The teachers has to spend necessary time on this, and they has to work through the difficulties they will meet some part of the distance learning how to use ICT tools actively in the different subjects. The activity have nothing to do with the operating system in use and has to be done independently of what system that's in use.

The most cost efficient solution is diskless workstation where all the software is maintained on one server for whole the school. The operating cost approximately half the price to operate compared to thin clients or standard workstations. It will require extra effort to build a network with good enough quality. In the beginning the building of the network is probably the most expensive part. After this is built the operational cost consists of around 60-70% of the total cost of ownership. To increase the competence in building and maintain a ICT solution really pays of. It does not be a big price difference on operating the system using external application provider compared to run it with municipality employees to run the technical part of the solution. We recommend an proactive maintenance strategy.

Avoid using graphical terminals as Free NX or Citrix on school wide installations. This kind of solutions is three times more expensive to operate than other solutions. Graphical terminals are nice as an unsupported alternative for pupils using the Skolelinux desktop at home. It is also used for teachers that uses an administrative applications from the municipalities administrative network. The networks are often divided to not mix restricted and personal information about citizens or pupils. Graphical terminals handles this nicely.

Generally there the suppliers has strong self interest when recommending client solutions to the schools. The functionally and the usability to the end user programs depends strongly on where the equipment is based concerning the bandwidth capacity to the schools. The main strategy should be to operating the systems centrally and place most of the equipment locally near the users. This will get the most efficient use of the bandwidth capacity and most functionality out for the clients.

When building up the ICT solution in the school there must be made a realistic budget. In Norway there is different standards for purchasing technical solution in general and special recommendations made for ICT installations in schools. This is not rocket science and it is relatively easy to get professional support. Other countries has also agencies that gives good recommendations for purchasing and rolling out ICT solutions in schools.

Decides if the pupil should save documents as OpenDocument or MS Word. When saving in OpenDocument the teachers has to be explained how to handle the relation to the home, and how it could be solved. Some municipalities support Free NX to the pupils at their home to families that have broadband. Others promote use of OpenOffice.org and gives all the pupils a live CD (Knoppix) with Windows and Linux versions of different software. Then the OpenDocument is not a big issue.

7. Debian: A force to be reckoned with

by Benjamin Mako Hill

Abstract:

This talk offers a ”Debian Themed” quick tour through the academic, legal, and business worlds. It overs insight into what everyone outside of Debian is saying about, doing with, and learning from the Debian project.

In doing so, it hopes to give Debian participants some insight into fields and areas that they are largely unfamiliar with (e.g., management, sociology, anthropology, economics, computer supported collaborative work, etc.). It illuminates what others – especially academics – find useful or inspiring about the project and to facilitate self-reflection and self-improvement within Debian. It reflects on the impact that Debian has had in the world beyond the Debian project and, in particular, in those areas that many Debian developers may not be familiar with.

A force to be reconed with

Over the past ten years, Debian has assembled an impressive list of achievements. Many of these achievements are immediately recognizable and intimately familiar to most Debian participants. These achievements, all made by the Debian project and *within the Debian project, include the fact that Debian has built up tens of thousands of packages, hundreds of derivatives, more than a thousand volunteers, a dozen quality releases, a handful of superb technologies and tools, and hundreds of thousands of fixed bugs.*

Certainly, Debian's has built an unparalleled distribution. Equally impressive, but more difficult to see from within the project, is the major effect that Debian has on other projects and in other realms. For example:

- *In academia, dozens of papers on Debian have been written and published from a variety of different perspectives. Researchers, some working within the Debian project and some completely detached, have unpacked the processes, motivations, and technologies that make Debian work.*
- *Legal scholars and license writers are increasingly aware and respectful of Debian in a variety of ways and the project continues to play an important role in the way that software and other forms of creative works are licensed and the way that licenses are written and revised.*
- *As people sit down to write software – free or not – they increasingly look to Debian as a model that can teach them a number of important things.*

While the analysis and conclusions in each of these areas paint a diverse and divergent set of pictures, together they demonstrate that Debian is a very interesting place to a large number of people for reasons that most of know nothing about. This talk aims to illuminate these areas.

Many of Debian's qualities and our achievements that are most interesting and important to others are overlooked or unknown within the project. In some cases, Debian simply takes things for granted. In others, developers' position in the center of the mix just makes things difficult to see. This talk aims to walk developers on a four-stop tour through Debian – as seen by those outside Debian or working outside of the distro-building world.

Through this tour, it aims to accomplish three goals:

1. To make Debian folks happy! This talk will provide Debian insiders with reasons to feel happy about their own achievements by demonstrating – with academic citations no less! – the impact that the project has had in a variety of fields. It will also illuminate the less favorable criticism. Luckily, the good feedback seems to outnumber the bad.
2. To provide in-depth insight into extra-Debian analysis of Debian so that, with added perspectives, we can better understand ourselves and our processes: our strengths and our weaknesses.
3. To make developers and others more conscious of their impact, their actions, and their strategies. Through this process, it hopes to prompt a self-reflective process where Debian can engage those outside the project in ways that build upon their knowledge to help us and to create mutually beneficial relationships.

The talk will focus in four areas where Debian has been studied or had major influence outside of the world distribution building:

- *Social Science and Collaborative Work: The talk will focus on academic literature of the social, ethical, and collaborative processes that make Debian successful. It will also touch on the academic literature on trust-building, mutual aid, and decision-making. It will introduce some of the conclusions and critiques leveled by researchers in these areas.*
- Software Engineering: This talk will pull from some of the work on software engineering techniques, review, quality, collaborative processes, and other areas that have been done with the Debian project as a subject or example. It will offer academic reviews and analysis of project processes and policies and suggestions that may be implicit in theses results.
- *Derivatives: Debian Derivatives that build on the work of Debian have clearly learned many things from Debian. Many of these are good things which they explicitly try to replicate in their own projects. Others are things they have moved away from Debian. Yet others are mistakes that Derivatives have made that Debian may want to avoid. This section will provide a whirlwind tour through these and will focus on my personal experience with the Ubuntu project.*
- Licenses: In the last year, Debian has played an important role in the revision processes of two major sets of licenses (the GNU GFDL and the Creative Commons licenses). The DFSG and Debian-legal have also played important roles in influencing thinking about free software licensing in the world beyond Debian. In the next year (as other talk proposals have alluded to) the FSF will be releasing the GPLv3. In this context, it is worth reflecting on the way that Debian has – and has not – been able to successfully influence licensing by third parties who want their software in Debian.

These four areas provide a good overview of four very different areas in which Debian has succeeded. This talk will build off my personal experience with Debian in many of these areas. For example, I have participated in academic research on Debian and have published three peer-reviewed articles about the project: two anthropological/sociological and one from the software engineering perspective. As a result, I am very familiar with the literature on Debian. Through my work with the Ubuntu project, I am also familiar with one of the most important Debian derivatives and what is has learned from and taken from Debian by derivatives. Finally, I have participated in both the GFDL and Creative Commons negotiation committees within Debian and have been following licensing issues, inside and outside Debian, for several years.

8. Debian's Debugging Debacle – The Debrief

by Erinn Clark and Anthony Towns

Introduction

This paper covers the topic of symbolic debugging of Debian applications, from both introductory concepts and techniques useful in debugging, to an overview of the underlying data formats and tools that enable symbolic debugging, finally pulling these concepts together in the context of Debian and examining how Debian can provide better symbolic debugging features for its users.

Debugging

Debugging is widely recognised as an unglamorous but essential part of software development. In *Code Complete*, Steve McConnell writes that "debugging is a make-or-break aspect of software development", and that "[while] the best approach is to [...] avoid errors in the first place [...] the difference between good and poor debugging performance is at least 10 to 1 and probably greater". Similarly, in *The Practice of Programming*, Brian Kernighan and Rob Pike write "Good programmers know that they spend as much time debugging as writing so they try to learn from their mistakes. Every bug you find can teach you how to prevent a similar bug from happening again or to recognize it if it does." and "Debugging is hard and can take long and unpredictable amounts of time, so the goal is to avoid having to do much of it."

But while debugging is an essential tool for software developers, one of the key advantages free software provides its users is the chance to join in the fun, and be able to actually fix things when they go wrong, instead of simply having to put up with or work around the misbehaviour. The network effects that arise from this form the foundation of one of the central tenets of Eric Raymond's paper *The Cathedral and the Bazaar*, "Given enough eyeballs, all bugs are shallow", dubbed Linus's Law.

A range of techniques fall under the general heading of debugging, not all of which require source code access. What they all have in common is an aim to allow the user to better understand what is actually happening when the program runs, as opposed to what was supposed to happen. There are a number of different approaches to gaining an understanding of what a program is doing – most break down into monitoring the program's internal behaviour, usually also slowing it down to make observation easier, or carefully observing the effects the program has on the system and inferring how the code works from that, or making use of the scientific method and making changes to the program or its input and seeing what effects that has.

The easiest form of debugging is often simply instrumenting the program itself, a practice often known as *printf debugging*, after the C function. By adding print statements at key places in your code, you can determine the values of various data structures, and see where the flow of control actually passes when the code is running. Often further tests will be added to verify assumptions automatically, such as by using C's *assert* macro. Since such instrumentation often makes a program run slower due to the fact it has more to display or simply more to calculate, these checks are often only enabled when a compile time flag is used.

The only disadvantage to printf debugging (as far as ease of use is concerned anyway) is that it requires some familiarity with the code in order to be useful – at the very least you need to know what the language's print command is, where in the code to put the printfs, and what data is likely to be interesting. This is often an inconvenience for users who don't have that familiarity – or more optimistically, who are trying to gain that familiarity – and in those cases, it can be more useful to instead treat the program as a black box, and examine how it interacts with other parts of the operating system and infer its implementation from its behaviour.

One of the most useful tools for this form of debugging is *strace*. As per its man page, strace "intercepts and records the system calls which are called by a process and the signals which are received by a process" – thus providing a precise transcript of the program's interaction the kernel, and thus the rest of the world. If printf debugging might be compared to determine what's going through someone's mind by asking them "what are you thinking?" every now and then, then strace is working out what someone thinks by watching their facial expressions and actions – harder to do accurately and requiring you to discard a lot of irrelevent information, but a lot less invasive, and generally more convenient. It's also something that works for proprietary programs that want more than just a penny for their thoughts.

The problem with strace is that it doesn't directly relate to the source code, so while you might be able to get hints as to what the program is doing wrong, or not doing at all that it should be, it may not be so helpful at presenting even

a general idea of where in the source code the problem is – obviously this lack is no problem for proprietary software when the source code is not available, but for free software we definitely want to be able to make the final leap from finding and understanding the problem to actually fixing it.

Symbolic debugging resolves this problem by keeping a link between the compiled code and its original source code by way of a symbol table. This provides tools such as *gdb* with the information they need to determine which line of source resulted in each segment of machine code, and the original variable names for the values the program stores in memory. By simply using the appropriate compiler option to include a symbol table with your executable, you allow users of your program to get the benefits of both printf and strace debugging at once – no prior familiarity with the code is required to start getting an idea of what is going on, and simply by firing up a debugger, you can delve into the program's control flow and data structures to whatever depth you might wish.

While the rest of the paper will talk about symbolic debugging, there are a range of other useful debugging techniques available too. These include tools to analyse memory usage such as *electric-fence* and *valgrind*, language specific debuggers such as *pydb* or *perl -d*, and other tracing tools such as *ltrace*, which investigates calls to shared libraries, and Solaris's *DTrace* which helps the administrator analyse behaviour of the entire system as well as just an individual program or process. Tools such as *xxgdb* and *ddd* provide alternative interfaces to symbolic debugging which can often be helpful as well.

Debian's Approach to Symbolic Debugging

As a binary distribution, Debian has chosen to sacrifice ease of debugging in favour of the smaller package sizes that is made possible by stripping symbol tables from our executables. The symbol tables for the X libraries add an extra 10MB to the normal installed size of 1MB, and an extra 2.6MB to download over the original 700kB, and this amount of increase is not particularly unusual.

Aside from the expected problems this causes, that is making it harder for users to understand what a program is doing and thus fix problems themselves, it also makes it harder for a developer to obtain useful information like a stack trace from a user-supplied coredump when debugging a crash, and it also makes it harder to do useful debugging of problems arising in libraries since the prospective debugger needs to recompile multiple packages just to obtain symbol information. Less immediately, not providing debugging information was one of a number of factors that led to the FSF ceasing to directly support Debian in April 1996.

Despite this, Debian undertakes a few approaches to help with symbolic debugging. In order to assist with recompilation, debian-policy recommends that packages check the *DEB_BUILD_OPTIONS* environment variable for a *nostrip* setting, and if found avoid stripping the symbol table from the final executables. This behaviour is handled automatically by most debian/rules toolkits such as *debhelper*.

Another approach, most commonly used for library packages, is to provide a separate debug package (usually called *libfoo-dbg*) containing the symbol tables for library packages that might be interesting to debug. This is very convenient for the user, because it allows debugging support to be enabled by a simple apt-get; but relatively inconvenient for the maintainer, because it increases the number of packages that need to be maintained for each architecture. As a result, there are under 200 debug packages, out of about 1500 library packages, and 18000 total packages.

A technique that also comes in useful, particularly if the only information available to debug a problem is a corefile, is to retain the original build tree used to make a package, and compare the corefile to the unstripped executable in the build tree, rather than the actual shipped binary – this is fairly difficult to arrange, since the original build tree will often not be available, particularly if it was built by a build daemon, but will work as long as the unstripped executable left in the build tree, was the one that the user was actually using.

Obviously, none of these techniques are terribly ideal – requiring extra effort from users and maintainers, or relying on being lucky enough that the problem occurs on a system that you've kept information around for. Fortunately, the above isn't all that's possible, and in particular, extending the scheme currently used for library debug packages seems entirely feasible.

ELFs and DWARFs

But before we can get into that in detail, some background on how executables and symbol tables actually work on Debian systems.

We begin with the executable format, which is used to arrange the various components that make up a program, including its code, hardcoded strings and data, information on linking the executable with other libraries, and anything else the compiler thinks is useful.

The standard executable format for Linux and many other Unix systems is ELF (Executable and Linkable Format). ELF is a common standard for executables (binary programs), object code, shared libraries, and core dumps. While ELF

provides enough of a standard to execute a program, however, it does not standardise the symbol tables for debugging information; for that we need DWARF (Debug With Arbitrary Record Format).

ELF is understood by a number of programs, probably the most important of which is the kernel itself: without being able to take an ELF executable and figure out how to load it into memory, the kernel would not be able to even link and run /sbin/init, let alone run anything else. At the other end of the spectrum, the compiler needs to be able to write out ELF objects or the programs it builds won't be able to be run.

In between those necessary extremes are a few tools that can manipulate ELF objects in useful ways. At the most basic level are tools such as *strip* and *objcopy* from the binutils package. As its name implies, strip is used primarily to strip symbols (such as the debug information) from object files, but it can also be used in the opposite sense – to strip everything but the debug information from the file. *objcopy* is a more powerful tool for manipulating ELF objects, which it does by copying a file and "translating" it in a variety of ways. A particularly useful translation is the *–add-gnu-debuglink* option, which allows you to take a stripped executable, and an ELF file containing the debug information that was stripped from it, and connect the two (which happens using a .gnu_debuglink ELF section).

This feature is only useful if other programs understand it, of course. We are particularly interested in whether it is understood by gdb, the GNU Debugger, initially developed by Richard Stallman in 1998, and which forms the basis for most debugging tools on free operating systems. Our hope is that when gdb tries debugging a stripped executable it will be clever enough to see our debuglink section, and realise the actual debugging symbols located elsewhere.

And, in fact, gdb is that clever – with some limitations. The most important of these is that the debug information needs to be located either in the same directory as the original executable, in a *.debug* subdirectory of that directory, or in a subdirectory of /usr/lib/debug that matches the path to the executable – this means that debug information for /usr/bin/foo would generally be in /usr/bin/.debug/foo or /usr/lib/debug/usr/bin/foo.

Red Hat uses a different set of tools for manipulating ELF files than binutils, namely *elfutils*. This suite offers fairly similar functionality in and of itself, but also provides a library used by a useful hack provided as part of the rpm package called *debugedit*. The useful feature debugedit provides is the ability to edit the information in debug symbols to indicate gdb should look for the source code in a different directory to that used to build the programs – so rather than expecting a random user to reconstruct the maintainer's favourite directory structure, the build directory can be canonicalised to somewhere under /usr/src.

Unfortunately elfutils is licensed under the *Open Software License*, which has been found in the past to not satisfy the Debian Free Software Guidelines, and to the best of the authors' knowledge no DFSG-free replacement for this functionality is currently available.

Debian Debugging FTW!

Putting this all together gives us a fairly straight-forward prospect for improving the user experience when trying to do symbolic debugging of Debian packages, while retaining most of the benefits of the current tradeoff, with respect to faster downloads and lower disk requirements for Debian systems. There are however a range of new tradeoffs to be made.

The basic plan then has to be to provide the separate, linked, debug information in a separate package – either a .deb following the existing -dbg tradition, or a variant packaging format – perhaps a simple tarball, or a reduced .deb variant similar to udebs. The tradeoff here is whether debugging packages, which should simply match the binary package it is installed with, should clutter up the Packages file, increasing the workload of apt and dpkg, or whether apt, dpkg, dak and similar tools could be augmented to support a new format for debug packages.

These debug packages could then either match individual binary packages, or could be combined to be one debug package per source – this choice probably doesn't make a major difference, with Debian having about 10,400 source packages, and 11,600 i386 binary packages. Once we've determined this, we need to make them as easy and automatic to generate as possible – ideally so that we don't require source modifications to packages, or NEW processing for each package.

Next we need to work out where source goes on users systems – the FHS suggests /usr/src – but we need some sort of directory hierarchy – eg /usr/src/debian/glibc-2.3.5-13. Once we've determined that, we also need to ensure that whatever tool downloads the debug information also grabs and unpacks the source – and we probably want to move towards the Wig and Pen source format to ensure that once the source is unpacked, it really is unpacked – gdb won't browse .tar.bz2 files very well.

Since we can't reasonably expect packages to be built under /usr/src, we also need some way to rewrite the debug location between when the package is built and when it's used. If elfutils were DFSG-free, this would not be a problem – we could expect developers and buildds to do this themselves; but without a DFSG-free tool, that's probably unreasonable. It wouldn't seem much better to ask users who want to do debug our software to install a non-free tool either; which doesn't leave many options. One would be to install the non-free software on ftp-master, and have it translate the

source code location in the debug information when including it in the archive. But is better debugging support worth running non-free code on one of our servers?

9. The X Community – History and Directions

by Keith Packard, Intel Open Source Technology Center, keith.packard@intel.com

Synopsis

This talk will present a social and political history of the X window system community and match that with contemporaneous technical changes to argue the thesis that the structure of free software governance is a key driver of the software development itself.

The Early Years: 1984 - 1988

Early X window system development, in the mid 1980's was run as a small software project at the MIT Laboratory of Computer Science. As with many university projects, the X project involved collaborations with other schools. Corporate involvement was limited to employees involved in university affiliated projects. Involvement from outside of the university environment was almost non existent.

At the time of X version 10, the Unix technical workstation market was led by Sun Microsystems. As there was no freely available window system at the time, Sun had developed a proprietary window system called SunView and independent software vendors were starting to write software for it.

The other Unix workstation vendors, still smarting from the success Sun had in pushing NFS in the Unix market, were not looking forward to being forced to adopt the SunView window system. A few people at these companies saw in the X window system an opportunity to break open the workstation market and force Sun to play by new rules.

A group of Digital Equipment employees convinced their management that by taking an improved version of the X window system and publishing it with a liberal license, they could appear to be setting a new shared standard for Unix window systems that would enable software developers to write applications that could be distributed on all Unix systems, rather than being limited to only Sun machines.

X version 10 was not ready to assume this role. Instead of small incremental changes as had been the prior practice, a completely new version of the protocol would be designed that would solve known issues and prepare the X window system for future technical challenges. As with any second-system, several large new features were added without having any real-world experience—external window management, synchronized event processing and the X selection mechanism to name a few.

The work to build version 11 was distributed with the existing collaborators, at MIT and elsewhere, working with a newly-formed team of Digital engineers in Palo Alto. While internet-based collaborative development seems natural for us now, at the time it was somewhat novel.

At the same time Version 11 development was going on, the limits of the original BSD TCP implementation were being amply demonstrated on the nascent internet. The BSD code responded to network overload by transmitting more packets. The effect on the internet at that time was to render it nearly inoperable for many months. As a result, the X window system developers took to late-night FTP sessions and eventually Fed-Ex packages full of mag tapes, which dramatically slowed down development.

The pain of a broken internet was felt deeply by the developers, and instead of acknowledging that the internet was now a critical development resource that just needed fixing, instead it was decided that distributed development was unworkable and that a centralized organization was needed where developers could work together.

The X Consortium

Starting with the assumption that centralized development was necessary, the Unix vendors constructed a new corporate consortium, housed at MIT and lead by the original X window system architect, Bob Scheifler. This consortium would hire engineers to work at MIT on the X window system and thus eliminate the internet as a limiting resource on progress. At its peak, the MIT X Consortium had 8 or 10 staff members, and an annual budget well in excess of a million dollars.

To feed this appetite, the X consortium sold memberships to major corporations who would pay for the privilege of controlling X window system development. An unanticipated consequence of this was that non-corporate involvement in the X window system was effectively cut-off. Projects at Stanford and CMU which had been making substantial

contributions to the community were now explicitly outside the mainline of X development, separated by a wall of money too high for any one individual to climb.

Another effect was that staff members of the X consortium were now more equal than other developers; only they had commit access to the source code repository, which left them as gate-keepers for the software. Corporations were discouraged from doing in-house developments as they were 'already paying' for X window system development through their consortium fees. The result was that a huge fraction of X11 development was done by MIT staff members.

The response to this increasing workload on MIT staff was to hire more staff. MIT offered a low overhead environment, and the consortium fees were high enough that money didn't seem to be an issue. Getting developers involved was reduced to a simple matter of paying them to sit at MIT. From a community software project spanning major universities and corporation around the world, the X window system had devolved to a corporate-directed contract programming house with offices on the MIT campus.

In hindsight, the resulting stagnation of the window system technology seems only too obvious.

Competing corporate interests eventually worked their way onto the X consortium board. With Sun admitting defeat in the window system wars, they were now busy re-targetting their SunView API to run on top of X and aligning themselves with AT&T by jointly producing a new toolkit called OpenLook. While the API was largely compatible with SunView, the look and feel were all new, and far better than anything else available at the time. But, it was closed-source.

Again, other Unix vendors didn't want to be forced to adopt Sun's proprietary technology, so they banded together to build another toolkit. Starting from the Xt toolkit intrinsics, they quickly cobbled together code from HP and Digital into the Motif toolkit. While the toolkit itself was badly designed and hard to use, it had one eye-catching feature—it simulated 3D effects on the screen by using a combination of light and dark colors on the edges of the objects.

Yes, bling beat out decent UI design. But, it wasn't really just bling. The 3D effect was used to market Motif as a more technically advanced toolkit, while in reality, it was just a shiny chrome finish on a steaming pile of thrown together code. Note that Motif itself was closed source at this point.

So, we had two toolkits, both closed source and both interested in becoming the standard for X development. The Xt intrinsics are theoretically capable of supporting multiple user interface designs, even within the same application. With this weak technical argument, and using the fact that they represented a majority of the X consortium membership, the Motif contingent was able to push the X consortium to adopt Xt as a standard.

With Xt as "the" intrinsics standard, AT&T and Sun were forced to create a whole new widget set using the OpenLook style based on the Xt intrinsics. The combination of a veneer of standardization of Motif (which remained closed source), broad (non-Sun/AT&T) vendor support and the fragmentation of the OpenLook market into three completely separate implementations (as the standard NeWS toolkit interface was also OpenLook), software vendors ran screaming from OpenLook to the "standard" Motif toolkit which then grew up to become CDE, the common desktop environment.

While the authors of the X11 specification were experts at networking and systems designs, they had no real expertise in fundamental 2D graphics. They built a primitive 2D rendering system from hearsay, speculation and a liberal interpretation of the PostScript red book. The results were sub-optimal on almost every axis. They were slow, ugly and of limited utility. As X11 included a sophisticated extension mechanism, the X11 authors assumed that one of the first extensions would be a complete replacement for the core X11 graphics functions. However, the corporate-controlled consortium was unable to muster any significant interest in this activity.

With a strong focus on conformance to the written standard, the graphics algorithms were rewritten to produce output that matched the specification. The resulting code was many times slower than the original approximate code, and had the unintended consequence of producing significantly less desirable results. No attempt was ever made to rewrite the obviously broken specifications to make them more usable or more efficient. That the spec was broken should have been obvious at the time—the portion of the specification involving stroking ellipses included both an eighth order polynomial and the exact computation of elliptic integrals. The former may be computed in closed form, but requires 256 bit arithmetic to produce the exact results required. The latter is not computable in closed form and can only be numerically approximated.

With 2D graphics now a settled issue, corporations turned their attentions to the 3rd dimension. SGI had developed the GL system for their workstations and was busy taking over the world of 3D technical computing. The remaining vendors were finding it difficult to compete, and decided to once-again try to reset the industry by replacing a corporate standard with an open one. The failure of the Programmers Hierarchical Interactive Graphics System (PHIGS) should serve as a warning to future groups attempting to do the same. While marginally better or worse open technology can replace closed technologies, it's all but impossible for bad open technology to do the same. However, it wasn't for lack of trying in this case. A huge group of companies got together to push PHIGS as a standard and spent huge amounts of money through the X consortium to develop a freely available implementation. Fortunately for us, the end users of these systems decided with their feet and we now have a freely available OpenGL implementation system for our use.

One of the last debacles of the X consortium was the X Imaging Extension. Digital had built a piece of hardware which incorporated sophisticated image processing hardware into a graphics accelerator. To get at the image processor, the system needed to synchronize access with the window system. The engineers decided to just build the image processing code right into the X server and let applications access it through a custom X extension. The group at Digital thought

this was a pretty cool idea and brought it to the X consortium with a view to standardizing it, probably with the hope that they'd somehow manage to generate demand for their image processing hardware. Image processing is a huge field, and while the group managed to restrict the problem space to "transport and display of digital images", the resulting extension design provided a complete programmable image processing pipeline within the X server. The sample implementation work was contracted out to the lowest bidder and the resulting code never managed to provide any significant utility.

The one tremendous success of the corporate consortium was the development of a comprehensive specification-based test suite. Throughout software development, the goal of every testing system is to completely and accurately test the software against a well designed specification. The X11 specification was well written, and the corporations were very interested in ensuring interoperability between various implementations. They contracted out work on a complete test suite to Unisoft, Ltd. who were experts in this kind of work. The resulting test suite was a marvel at the time, and remains invaluable today. The number of specification clarifications and minor code revisions needed to support the test suite were not insignificant, and the result was a code base which passed the test suite. With the exception of wide arcs.

So we see that corporations use the mechanism of an industrial consortium to push their own agenda, at the expense of their competitors, and without regard to technical capability. None of this should be at all surprising, nor is it really wrong from the perspective of the corporate owners.

Of course, we all know the end result of this fractious Unix in-fighting—Microsoft grew up from the bottom and took over the entire desktop market. Large corporations aren't known for their long-term vision...

With the death of the Unix technical workstation market, the X consortium grew moribund; while the workstation market wasn't growing any longer, it didn't entirely disappear. However, with ever-shrinking revenue streams, the Unix vendors grew less and less interested in new technology and more and more interested in avoiding additional expenditures. In 1997, the X consortium was re-parented under the Open Group who looked upon it as a potential licensing revenue stream. Attempts by TOG to extract money from this formerly lucrative franchise proved futile and in 2004 the X Consortium was partially separated from TOG by the member companies and limped along with a modest budget and strict instructions to not change anything to radically. A few feeble attempts to jump-start development within that structure went for naught.

XFree86

Around 1990, Thomas Roell from the Technische Universität München started developing X drivers for several of the early PC graphics cards for use under various PC-based commercial Unix variants. While the graphics capabilities of Unix workstations were fairly reasonable, these early PC systems were primitive, offering low resolution and even lower performance. Thomas released this work under the MIT license as X386 1.1. Without any consortium members supporting this effort, the X consortium staff, this author included, largely ignored this work. Thomas eventually hooked up with Mark Snitily and they worked together to improve the code.

To get X386 included in the MIT release, Mark and Thomas needed some way to work with the X Consortium. As the Consortium membership was strictly limited to corporations, Mark's consulting company, SGCS joined the Consortium and they lobbied to have the X386 code included in the release. The resulting X386 1.2 was released, still under the MIT license, with X11R5. That code was not entirely functional at release time and served mostly as a base for future development than a usable PC-based X server.

To continue to support the now-necessary consulting company, Mark and Thomas decided to commercialize their work and created a product based on the X386 code, improving and enhancing it. Those improvements were not pushed back into the X consortium code base and were instead treated as part of a strictly closed-source product. What had started as a generous offer to the community to create an open source PC-based window system now became just another commercial product. The consulting company eventually turned into Xi Graphics and still exists in a small niche selling closed-source X servers.

Around the same time, David Dawes, David Wexelblat, Glenn Lai and Jim Tsillas were running various Unix variants on their Intel-based machines. They started collaborating to improve the support for X in this environment. Patches and eventually a complete driver suite were passed around, with the intent that they be plugged into the standard X consortium release and built in situ.

The "gang of four" eventually released an extended version of the code and called it X386 1.2e. As SGCS was now selling a commercial closed-source product called X386, this new group was forced to use a different name. They decided to call the project XFree86, a pun on the X386 name which also made clear their intentions to keep the software freely available.

Again, to gain some visibility in the X consortium, this group needed a corporate voice. They created a company, The XFree86 Project, Inc. and received a donation that allowed them to buy a consortium membership. Expecting a warm welcome by the X consortium staff, they were disappointed to find that without a 'real' company behind them, the consortium dismissed their efforts.

Because the XFree86 Project, Inc. was created entirely to satisfy the requirements for membership in the X Consortium, the governance structure was made as light-weight as possible. The original developers would be the only members and would elect officers from their number. Not an unusual structure for a small company. By 2002 all but one of the original XFree86 founders had departed the project for other activities. Governance was left in the hands of the remaining original member. The other founders were still members, but without active interest in the project, their involvement in project oversight was minimal.

In any case, these developers' experience with the corporate-controlled X consortium left them with a strong desire to avoid corporate control of XFree86.

Because of the commitment to support an open source version of the window system, the XFree86 project grew to support most of the Intel-based unices, and, in 1992, it was running on GNU/Linux (more changes were required in Linux than XFree86 to make this work). A healthy community of developers worked to ensure that the system ran on each operating system and supported the available video cards.

In 1998, the Open Group attempted to change the X license to permit them to extract license fees for its use, the XFree86 project refused to accede and said that they'd continue to support the previous version of the window system for use on Intel-based Unix systems. This display of brinksmanship succeeded, and TOG was forced to undo their license change and X11R6.4 was released with the old MIT license (although TOG licenses from that period can still be found in some files).

The eventual result of this battle was the ascension of XFree86 as the new home for X development. However, the core of the XFree86 development efforts had always been in building graphics drivers for the existing window system. There was no professed interest from the XFree86 leadership in taking over this significantly larger role.

The XFree86 project made some important contributions to the code base, most notably the development of the loadable X server, which was aided by significant donations of code from Metrolink. In the loadable server, individual video and input drivers along with font loading systems and other pieces could be plugged into the X server at run time. Before that architecture was developed, each family of graphics cards had a separate X server executable, and there was no support for adding new input drivers. In addition, XFree86 developers designed a highly optimized rendering architecture which maximized performance of the core X rendering operations on cards of the era. Advances in graphics hardware may have made this old acceleration architecture obsolete, but it was extremely important at the time it was developed.

The XFree86 project also accepted contributions from non-members that extended the window system in some interesting new ways: video overlays, Unicode support in applications and libraries and new rendering extensions. Most of these contributions remain useful today. The result was a window system which went far beyond that provided by the X consortium.

Many of the XFree86 contributors wanted to see XFree86 accept the mantle of overall X development, but were unable to convince the project leaders to take on this additional responsibility.

As the XFree86 membership is fixed to the original founders, there was no way for new contributors to gain any political influence within the organization. The few remaining active founders saw that the new contributions to the window system were coming from people supported in their work by large corporations and feared that changes in the structure of the organization might lead back to corporate control over the window system. Eventually, communication broke down between the founders and the non-member contributors and the non-members were ejected from participation in the project. Without any way to change the project from within, these contributors were left without a place to do their work.

The X.Org Foundation

Meanwhile, GNU/Linux had spread out from its roots and was starting to make inroads into the Unix workstation market, and in some minor areas of the Windows market as well. The former Unix vendors, along with new GNU/Linux vendors and the disenfranchised XFree86 contributors were all actively working separately on X technologies. Eventually, these three groups found each other and decided to "fix" the X consortium.

The existing X.Org Consortium was reconstructed as the X.Org Foundation. As a charitable educational foundation, the X.Org Foundation is a target for tax-exempt donations. Corporations with an interest in advancing the X window system are encouraged to sponsor the organization with donations.

There are several important differences between the old Consortium structure and the new Foundation structure. The first is that governance of the organization is entirely separate from any financial contributions made by sponsoring corporations. The second is that membership is on an individual basis, and open to anyone making substantive contributions to the window system. As with many other organizations, the membership elect a board of directors who are responsible for technical oversight and corporate management of the organization.

The sponsoring corporations form a separate sponsor's board within the organization which is responsible for guiding the Board's financial decisions. So far, this has mostly meant that the X.Org foundation isn't spending significant amounts of money as no strong consensus over how the monies would be spent most effectively. The old X.Org Consortium spent

most of their budget on development efforts, either hiring in-house developers or contracting various projects out to separate companies. The new Foundation board has a very strong bias against paying for work which should be done directly by the people with an interest in seeing that work get done, either companies or individuals.

And, the Board hasn't found itself a strong voice in the community yet, leaving the day-to-day technical details to the self-appointed maintainers of individual modules within the code base. One big disagreement did flare up in the 6.8 release process when changes to the ancient Athena toolkit to support the controversial XPrint extension were incorporated into the tree. To forestall a long and bitter dispute, the parties agreed to let the Board adjudicate. Unfortunately, democratic systems don't always produce the best results, and XPrint support in the Athena toolkit was included in the 6.8 release.

With acceptance of overall X window system leadership, the X.Org Foundation has started to push significant changes into the code, leading to some interesting new capabilities in the window system and, perhaps, a greater use of GNU/Linux for desktop computing throughout the world.

Dictatorships are Efficient

In this brief trip through X history, the connection between governance and development is evident at several points. The original project, guided by Bob Scheifler as a benevolent dictator saw rapid development and wide-spread exploration into the technology. Major universities were directly involved in the project and were basing course-work around the window system. Large corporations were starting to build and deliver X-based products. Small contributors were able to be heard and have their ideas evaluated by the group. In short, this period represents the golden age of X development where politics took a back seat to technology as one trusted person was capable of making the necessary decisions and seeing them broadly accepted by the whole community. The X protocol itself evolved in incompatible ways 8 times in the first four years, demonstrating a willingness to take risks and break everything.

This era wasn't perfect; major structural changes were needed in the window system for it to be usable in a larger spectrum of environments. Mechanisms, like sophisticated user interfaces for window management, commonly used in other window system environments were difficult or impossible to replicate in X. Getting enough people involved in a large scale re-engineering effort wasn't feasible given the resources available.

The X11 development effort was funded by Digital for entirely selfish reasons; convincing the Unix workstation industry to switch from Sun's proprietary window system to X would mean that the built-in advantage that Sun enjoyed in the market would be wiped out, leaving other workstation vendors free to compete on a more even footing. However, once that result had been assured, Digital management weren't interested in continuing to fund development at that level.

Corporations are Hidebound

The abrupt migration from benevolent dictator to corporate consortium was caused in large part by the failure of the nascent internet to fully support the distributed development model which produced X11. The resulting creation of a centralized development team funded by a corporate consortium brought about radical changes in the scope and direction of the work included in X releases. Gone were the rapid advances in technology. Gone too was the active and involved participation of universities in the project. Even Project Athena at MIT was suddenly far less able to influence the development of the window system.

In their place was a strong focus on standardization, testing, performance and documentation. The X window system was even put up for ANSI standardization, an effort which still haunts the memory of this author. The performance work lead to significant improvements in the speed of the code, although any developer who has seen cfb8line.c may question the development methodology used. The death of the Unix workstation market eventually made the organization irrelevant.

Private Clubs are Limited

The XFree86 Project, Inc. was built to be a member in the X consortium. The founders had a simple focus on ensuring the existing X window system ran well on PC class hardware. In this area they excelled. XFree86 became the standard window system for all free software systems and had numerous successes with hardware vendors in getting a range of specifications, open source drivers or compatible closed-source drivers for virtually all available graphics hardware. Their steadfast commitment to open source kept the window system from falling under the Open Group's attempts to leverage their copyright into piles of money.

However, with its fixed membership, it was unable to adapt to changes in the contributor population which happens in every open source project. New contributor constituencies were unable to effect a change in project direction leading to dissatisfaction and an eventual falling out with no end of bruised egos left as a result.

Foundations are Functional

With wide contributor buy-in, and a lassez faire Board, the X.Org foundation is currently managing to satisfy the range of contributor interests and push the X window system in some new and interesting areas. Its foundation as a non-profit organization and limited financial activity have caused the old pay-for-say policies to disappear leaving the more relevant contribute-for-say policy dominant.

So far, the X.Org Foundation has steered a careful course between competing interests and has managed to finally unify the majority of contributors under one organizational banner. We'll see how well this democratic system fairs in the future as the GNU/Linux desktop continues on its path to total world domination.

10. Lightning Talks

Joey Hess et al

Abstract

Lightning talks are a way to let a variety of people speak on a variety of subjects, without a lot of formal conference overhead. Each talk is limited to 5 minutes, and the talks are presented back-to-back throughout the 45 minute session. Just as a one-liner can be interesting and useful despite its short length, the five minutes of a lightning talk is just enough time to discuss the core of an idea, technique, interesting peice of software, etc. And of course, if one of the talks isn't interesting, another will be along in just five minutes.

10.1. Tenative lightning talk schedule

time	speaker	title
2 minutes	Joey Hess	introduction
5 minutes	Jeroen van Wolffelaar	Actively discovering bugs/issues with packages
5 minutes	Jose Parrella	Walkthrough: Make your Country love Debian
5 minutes	Andreas Schuldei	Debian in the greater Linux ecosystem
5 minutes	David Moreno Garza	WNPP: Automating the unautomatizable
5 minutes	Raphael Hertzog	How far can we go with a collaborative maintenance infrastructure
5 minutes	Matt Taggart	How to get debian-admin to help you
5 minutes	Joey Hess	Significant Choices
2 minutes	Jeroen van Wolffelaar	Tracking MIA developers
3 minutes	Jeroen van Wolffelaar	Datamining on Debian packages metadata

Alternates

time	speaker	title
5 minutes	Joey Hess	Debian in 4 MB or less
2+ minutes	Jeroen van Wolffelaar	How to pronounce Jeroen van Wolffelaar, and other names

10.2. Talk summaries

10.2.1. Introduction

(Presenters, line up!)

Just explaining what a lightning talk is and how things will work.

Basically, here is a fixed microphone into which you will start by giving your name and the talk title, here is a video hookup, here is a noisemaker that I will sound after your 5 minutes are up at which point you are DONE and it's the next person's turn. Have fun!

Presented by Joey Hess

10.2.2. Actively discovering bugs/issues with packages

There are several tools to check a package automatically: lintian, piuparts, building with pbuilder. How to do execute those continuously, and especially, make the results immediately available to all, and have issues directly reported to the maintainers.

Presented by Jeroen van Wolffelaar

10.2.3. Walkthrough: Make your Country love Debian

A short tale about the Venezuelan experience in the Free Software Migration, and how each day it is turning strongly towards Debian GNU/Linux. Including Government migration towards Debian, people using Debian as free, democratic Internet access platform, and Debian community activities.

Presented by Jose Parrella

10.2.4. Debian in the greater Linux ecosystem

I would like to pass on what I hear from sponsors/supporters of Debian when talking to them. This includes e.g. Debian's role in the LSB landscape, its perception by some service selling companies, and directions it could take when placing itself in the market in the future.

Presented by Andreas Schuldei

10.2.5. WNPP: Automatizing the unautomatizable

Let's talk about what's been done on the WNPP field by all the people involved on it. What can be improved or what can be implemented to make things easier for WNPP maintainers.

Presented by David Moreno Garza

10.2.6. How far can we go with a collaborative maintenance infrastructure

Short presentation of the Collaborative Maintenance proposal and possible implications that it can have on NM, QA, and our way to maintain the packages. Explain how that infrastructure fits with the PTS and everything else.

Presented by Raphael Hertzog

10.2.7. How to get debian-admin to help you

The way in which you craft a request to debian-admin has a great affect on how quickly they can help you. This talk will help you determine what things debian-admin will be able to help you with and what information to include in the request to order to get help quickly without the need for extra questions.

Presented by Matt Taggart

10.2.8. Significant Choices

A rant on how some decisions are made in Debian in less than ideal ways and the surprising consequences.

Presented by Joey Hess

10.2.9. Tracking MIA developers

A brief rundown of the infrastructure, but more importantly procedures and customs, used to find and take action on people who are MIA, or more accurately put, people who are suspected of having themselves overcommitted to Debian work.

Presented by: Jeroen van Wolffelaar

10.2.10. Datamining on Debian packages metadata

There is a lot of data available about Debian packages. $>$ 10G of bug data, packages files, upload logs, who sponsors who, and dozens of other sources. They are hard to present and correllate though: an identity isn't clearly defined (people can have multiple gpg keys, email addresses, names, and some of those can even clash). This is where carnivore comes in, a new QA tool to assist here. Also discussing other techniques and applications.

Presented by Jeroen van Wolffelaar

10.2.11. Debian in 4 MB or less

Explaining how the ADS root builder can create embedded systems based on Debian and how this can tie in with projects like Debonaras.

Presented by Joey Hess

10.2.12. How to pronounce Jeroen van Wolffelaar, and other names

”Random-J”, ”Jeroen van Wifflepuck”, all not correct... Short introduction to Dutch sounds not found in English or most other languages.

Presented by Jeroen van Wolffelaaaaar

Part II.

Workshops

11. Weeding out security bugs

by Javier Fernandez-Sanguino

How do security bugs affect the Debian project

The Debian project asserts that it provides a high-quality distribution (Debian GNU/Linux) with a release cycle that is not forced upon by marketing requirements and, consequently, makes it possible to provide a distribution without important (i.e. release critical) defects.

However, once a release is done, any security bug affecting a software package that is part of the release has many direct consequences:

- our users' systems are immediately in danger of being compromised due to the security bugs (this will depend on the nature of the bug itself, and whether it's a local or remote exploitable bug)
- our security team needs to deal with the security bug in order to provide a new fixed software package backporting, or writing themselves, a patch fixing the security bugs
- when the fix is developed our buildd infrastructure needs to handle the fix and generate new packages in short time
- when an advisory is sent, after a new package version is available with the fix in our security servers, our security support infrastructure (bandwidth and services) has to cope with hundreds of users downloading the new version of the package to install the upgrade
- even if the security bug is fixed, there is always the possibility that the fix or the changes in the package introduce new bugs that will affect our users (even though they may not be security related)

If any of these steps fail and, consequently, the "window of exposure" (time it takes from a security vulnerability to be known to a patch be available by us) increases then this impacts negatively in the project, new sites will pick this up and it will become bad publicity.

Security bugs have a negative impact even if the our patching process works out flawlessly: we are able to produce patches in time for all our supported architectures (or even before the vulnerability is publicly known) and there are no hiccups with any of our infrastructure. When doing a review of the number of security bugs found for a given release, reviewers might find that the release process has not been adequate if the bugs found **after** the release is too high. Indeed, our release process was designed partly to find (and fix) these kind of bugs, if there are too many advisories published after a release then that might be an indication that there is a flaw in our release process.

There is also the issue of quantity, regardless of the previous issues, an increasing number of security bugs require an increasing number of resources from the Debian project. These resources increase: CPU (in different architectures to drive the security buildds), bandwidth (for the download of the patches), and, most important, human (the people that have to develop the patch, test it and write the advisory).

Security issues in the Debian distribution

The Debian Security Team has issued (since 2001 and up to April the 5th 2006) 1017 advisories for 1749 distinct vulnerabilities. Of these, over 60% have been related to remote vulnerabilities. This is not necessarily the **real** distribution of vulnerabilies of the different releases the project, it is the number of vulnerabilities that the project has issued advisories for.

This is the list of classes of security bugs found in Debian packages[1] as well as the percentage of vulnerabilities fixed in issued advisories:

buffer overflows the input being received by a system, be it human or machine generated, causes the system to exceed an assumed boundary. This might be considered a subset of improper data handling, but the large number of applications and the consequences of this bug (code execution) justify it being considered a different class of bug (almost 27% of security vulnerabilities);

[1] It is based on the published DSAs crossed with the information available in the National Vulnerability Database `http://nvd.nist.gov/` (NVD, formerly ICAT) based on the CVE name of vulnerabilities.

improper data input handling the input being received by a system is not properly checked such that a vulnerability is present that can be exploited by a certain input sequence. This issue leads to many type of different attacks, such as cross-site scripting in web applications, or SQL injection (almost 25% of security vulnerabilities);

design errors when there does not exists errors in the implementation or configuration of a system, but the initial design causes a vulnerability to exist (18,7%);

boundary condition error the input being received by a system, be it human or machine generated, causes the system to exceed an assumed boundary. It is also a subset of input validation (7%);

exceptional condition handling handling (or mishandling) of the exception by the system that enables a vulnerability (6,5%);

access validation error the access control mechanism is faulty (4,7%);

race conditions the non-atomicity of a security check causes the existence of a vulnerability (2,6%);

configuration error user controllable settings in a system are set such that the system is vulnerable (2,4%);

environmental error: the environment in which a system is installed somehow causes the system to be vulnerable (0,9%).

All these bugs are, in themselves, defects in the software itself. An application that fails to validate input[2] coming from untrusted users[3] might introduce a security vulnerability which can range from a buffer overflow remotely exploitable in a server daemon to a SQL injection error in a web-driven application.

The following is the number of advisories (and vulnerabilities) for the different distributions Debian has released[4]:

- 197 advisories for 256 vulnerabilities were published for Debian 2.2 (*potato*) which was in security maintenance for 2.79 years. There are 59 million lines of source code in this release;
- 690 advisories for 1070 vulnerabilities have been published for Debian 3.0 (*woody*) which has been in security maintenance for 3.7 years. There are 105 million lines of source code in this release;
- 271 advisories 570 vulnerabilities have been published for Debian 3.1 (*sarge*) in less than a year. This release has 216 million lines of source code.

Nobody will be surprised when told that the number of security vulnerabilities (and, consequently, advisories) published for an operating system is very dependant on the amount of software it includes, more software means more bugs. A recent analysis by Coverity[5], a company that provides a closed-source source code audit software, shows an average of 0.3 defects per thousand lines of code for some of the most popular and used FLOSS projects. Not all of these defects might be *exploitable* security bugs, but the more the distribution grows[6] the more security bugs it will hold.

It is important for Debian developers to know and understand the different types of vulnerabilities as well as to know what they could have done to prevent a programming bug to become a security issue. This includes: designing servers so that they properly implement privilege separation instead of running as root, avoiding the use of setuid or setgid binaries and providing good installation defaults such as not starting up a service if it is not properly configured or limiting access to an application to only the server it is installed on.

Work of the Debian security audit team

The Debian Security Audit Team `http://www.debian.org/security/audit/` started working in 2004 to focus work on auditing Debian packages for security issues. It has been direct responsible of 82 Debian Security Advisories and has opened up 122 security-related bugs in the BTS (up to march 2006).

The Audit Team is composed of loosely coordinated group of people. Although they use a public mailing list, more of the audit work is "hidden" and is not even discussed on list until an advisory is published. Currently, the different

[2] For more information see the "Validate All Input" section of the David Wheeler's Secure Programming for Linux and Unix HOWTO `http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/input.html`.

[3] In these case they can be either remote users, for daemons, or local users for setuid/setgid applications.

[4] I have also include the size of the distribution in millions lines of source code based on the Libre software engineering (Libresoft) research group from the Universidad Rey Juan Carlos, as detailed in Debian Counting `http://libresoft.dat.escet.urjc.es/debian-counting/`.

[5] For more information see Automating and Accelerating Open Source Quality `http://scan.coverity.com/`, an analysis of thirty open source projects including the Linux kernel, gcc, FreeBSD, NetBSD, Apache, Samba, Perl, Firefox and GNOME. LWN coverage (with interesting discussion) is at http://lwn.net/Articles/174426/

[6] And based on Libresoft's data it is currently doubling its size every two years!

members of the Audit Team focus on one specific type of bug and work their way through the package sources in order to find instances of that type of security bug.

One of the goals of the Audit Team is to have security bugs fixed in the distribution before they are really an issue (i.e. before the affected package versions are released).

Occasionally, members of the team also review security bugs and advisories from other distributions and make sure that the Debian package that provides the same software is fixed in Debian too. At times, this overlaps with the work already done by the Stable and Testing Security Teams but it often means that there are more "eyes" looking for (known) security bugs that might be present in the software we distribution.

These are some of the lessons learned by the team:

- many developers are not aware of the consequences of some security bugs and need to be shown that a security bug is of higher severity;
- even though some bugs have been found and reported, there are many more *security* bugs present waiting to be removed. This specially applies to software that is not too popular (consequently, not many people are looking for bugs in it) or security type of bugs that are not being often reviewed;
- there is too much software in the distribution and auditing resources are scarce;
- the available free software tools for source code review are insufficient for the task at hand;
- it takes quite some time to fix security bugs. Specially security bugs which are not highly critical (such as temporary file vulnerabilities). This is related to the limited resources of the Debian Security team but it also happens because of maintainers being unresponsive.

How can a developer improve security in the Debian OS

When you are packaging software for other users you should make a best effort to ensure that the installation of the software, or its use, does not introduce security risks to either the system it is installed on or its users.

You should make your best to review the source code of the package and detect issues that might introduce security bugs before the software is released with the distribution. The programming bugs which lead to security bugs typically include: buffer overflows `http://en.wikipedia.org/wiki/Buffer_overflow`, format string overflows, heap overflows, integer overflows (in C/C++ programs), and temporary symlink race conditions `http://en.wikipedia.org/wiki/Symlink_race`(very common in Shell scripts).

Some of these issues might not be easy to spot unless you are an expert in the programming language the program uses, but some security problems are easy to detect and fix. For example, finding temporary race conditions in source code can easily be done by just running `grep -r "/tmp/" .` in the source code and replace hard coded filenames using temporary directories to calls to either *mktemp* or *tempfile* in Shell scripts, or File::Temp in Perl scripts, and *tmpfile* in C/C++ code. You can also use source code audit tools [7] to assist to the security code review phase.

When packaging software make sure that:

- It is not alpha or beta software, if it is, prevent it from going into *testing* (by introducing an RC bug for it). If it's not ready for release, don't let it be released.
- The software runs with the minimum privileges it needs. That is:

1. the package does install binaries setuid or setgid[8];
2. if the package provides a service, the daemons installed should run as a low privileged user, not as root.

- Programmed periodic tasks (i.e., *cron*) installed in the system do not run as root or, if they do, do not implement complex tasks.
- The default configuration is sane and limits exposure. Don't think that everybody will install the software in a development enviroment and needs all the bells and whistles the program might provide.

If you are packaging software that has to run with root privileges or introduces tasks that run as root, make really sure it has been audited for security bugs upstream. If you are not sure about the software you are packaging, or need help, you can contact the Debian Security Audit team and ask for a review. In the case of setuid/setgid binaries, you must

[7] More information available at http://www.debian.org/security/audit/tools

[8] *Lintian* will warn of setuid, setgid binaries in the package

follow the Debian policy section on permissions and owners http://www.debian.org/doc/debian-policy/ch-files.html#s10.9.

Once your software has been released, make sure that you track security bugs affecting your packages either through upstream mailing lists or through security mailing lists. If a security bug is detected that affects your package you must follow the Handling security-related bugs http://www.debian.org/doc/manuals/developers-reference/ch-pkgs.en.html#s-bug-security guidelines in the Developer's reference. Basically this boils down to contacting the security team to let them know, and help produce (and test) patches for the software versions released.

Finally, invest time in reading about security bugs and how to prevent them. David Wheeler's Secure Programming for Linux and Unix HOWTO http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html should be a must read, it is an online book freely available packed full of valuable content. For developers that package web-based applications, the OWASP Guide http://www.owasp.org/documentation/guide.html is also a must read. Other recommended reading would be Secure Coding: Principles —& Practices http://www.securecoding.org , by Mark G. Graff and Kenneth R.Van Wyk (ISBN 0596002424)

Conclusion

The constant growth of the distribution makes it inevitable to have a large number of security bugs in it. Security bugs drain important resources from the project but developers have it in their own hands to improve the situation by making sure they provide releasable software and they prevent software that might not be releasable (unaudited, alpha or beta software) from getting into the distribution.on.

Security safeguards might be introduced in the distribution, such as stack overflow prevention measures (as implemented in OpenBSD or Adamantix) or Mandatory Access Control mechanisms (such as SElinux). But these safeguards will only protect our users against specific set of attacks, users cannot (and should not) rely on them to protect their systems against every possible instance of security bugs.

Also, unfortunately, due to the current status of automatic source code audit tools it is not possible, for the moment, to design or provide something akin to lintian.debian.org to warn Debian developers (and users) of possible security bugs in Debian packages. We are currently missing metrics to evaluate the quality (security-wise) of Debian packages (and the software they include) to both detect and make decisions about software distributed within Debian.

That makes developer awareness on information security issues something even more important if we want to be successful in providing a high-quality universal operating system.

12. Debian Installer Internals

by Frans Pop <fjp@debian.org>

Abstract

The Debian Installer is sometimes described as a mini Linux distribution which gives an indication of its complexity. This paper gives an introduction to the inner workings of the installer when it is running, its components (udebs) and its build system.

Running Debian Installer

The examples in this section reflect the current status of the Installer, the Etch Beta 2 release, and are based on the CD-ROM and netboot installation methods for i386. The choice for i386 was made as this is most familiar to most users, but installations for other architectures is not structurally different[1].

So what are the basic steps when the installer is run? In any installation five stages can be distinguished.

1. boot and initialization - setting up the installer so that it can load any additional components
2. loading additional components - expanding the installer to its full functionality)
3. network configuration (unless already done in stage 1)
4. partitioning
5. installing the target system

The first three stages are where the fundamental difference between installation methods can be seen. All components (udebs) used there need to be included in the initrd[2] with which the installer is booted.

The table below shows what components are involved in the first and second stage for the CD-ROM and netboot installation methods and also shows where these differ.

Stage	CD-ROM	NETBOOT	Comments
-	initrd-preseed		Only if /preseed.cfg is present
1	localechooser		Language/country/locale selection
1	kbd-chooser		Keyboard selection
1	cdrom-detect	eth-detect	Hardware detection and setup
1		netcfg	Network configuration
-	file-preseed	network-preseed	If selected at boot prompt
2		choose-mirror	Mirror selection
2	load-cdrom (anna)	download-installer (anna)	Retrieve and unpack additional components
3	eth-detect		Hardware detection and setup
3	netcfg		Network configuration
3	choose-mirror		Mirror selection (sometimes needed for stage 4)

[1] Some architectures currently do not use partman for partitioning, file system creation and mount point selection but instead use other components. Some architectures also use specific additional components as part of their default installation. However, this does not make them structurally different.

[2] With one exception. In floppy disk based installations, the initrd does not contain all needed components for stage 2, but it does have the ability to load additional components that belong to stage one or are needed for stage 2 from additional floppy disks.

The remainder of the installation is basically the same for all installation methods.

Stage		Comments
4	hw-detect	Additional hardware detection
4	partman	Partitioning, file system creation and mount point selection
5	tzsetup	Time zone selection
5	clock-setup	Configure for hardware clock set to UTC or local time zone
5	user-setup	Set up root and normal user accounts
5	base-installer	Base system (debootstrap) & kernel installation
5	apt-setup	APT configuration in the target system (sources.list)
5	pkgsel	Select and install additional packages (tasksel)
5	grub/lilo-installer; nobootloader	Boot loader installation
5	prebaseconfig[a)]	Finish up the installation and reboot

[a)] Will soon be renamed to finish-install. The name "prebaseconfig" no longer makes any sense as base-config was obsoleted with Etch Beta 2.

Installation methods

The installer supports a lot of different installation methods and in some cases installation methods can be creatively combined. The definition of an installation method is based on the following questions.

- How is the installer booted?
- From where are additional udebs retrieved?
- From where are packages needed to install the base system retrieved?
- From where are packages needed to install tasks retrieved[3]?

The table below shows the answers to these questions for the most common installation methods.

Method	Boot	Udebs	Base system	Tasks
netboot	network (TFTP server)	network	network	network
mini.iso	CD	network	network	network
businesscard CD	CD	CD	network	network
netinst CD	CD	CD	CD	network
full CD/DVD	CD	CD	CD	CD (+ network)
hd-media[b)]	harddisk/USB stick	CD image	CD image/network	CD image/network
floppy (net)	boot/root/net-drivers	network	network	network
floppy (cd)[b)]	boot/root/cd-drivers	CD	CD/network	CD/network

[b)] Whether packages for the base system and tasks are retrieved from CD (image) or the network depends on the type of CD used in combination with these boot methods.

The boot process

The boot process for the installer is similar to the boot of a regular system. A bootloader (in some cases the system's firmware) is responsible for loading the kernel and loading the initrd after which init is started. The boot process can be debugged by adding the BOOT_DEBUG parameter.

It is possible to pass additional kernel and boot parameters. Kernel parameters are sometimes needed to get non-conformant hardware supported, or to install from serial console instead of an attached keyboard/display.

Boot parameters can also be used to influence the installer itself. More about this in the section on preseeding.

The boot process is currently quite complex as it needs to support several generations of linux:

- 2.2, 2.4 and 2.6 kernels with devfs, all subtly different
- 2.6 kernels with udev; several generations, both with and without hotplug

[3] The last question can also be rephrased as "what source lines are set up in the /etc/apt/sources.list file for the target system".

The following enumeration gives an overview of the main steps in the boot process.

1. `/init` (initramfs) or `/sbin/init` (initrd)
   Sets up initial environment (/proc, /dev; run udev)
2. busybox init
   /etc/inittab is parsed; this contains:
   - ::sysinit:/sbin/debian-installer-startup
   - ::respawn:/sbin/debian-installer
   - init for VT2 (busybox shell), VT3 (/var/log/messages), VT4 (/var/log/syslog)
3. /sbin/debian-installer-startup
   This is a run-parts like script that executes or sources scripts in `/lib/debian-installer-startup.d`. The main functions that are performed are:
   - mount devfs and sysfs if appropriate
   - run udev/hotplug
   - load acpi modules
   - start syslog
   - check available memory and, depending on preset limits, enable a lowmem mode if needed
   - load any debconf templates present in the initrd
   - parse boot parameters to look for debconf patterns (<component>/<template>=<value>) and set these in the debconf database
   - detect the type of console that is used
   - load framebuffer modules
   - detect if the installer should run in rescue mode
4. /sbin/debian-installer
   This is a run-parts like script that sources scripts in `/lib/debian-installer.d`. The main functions that are performed are:
   - detect if a framebuffer device is available
   - initialize the console (blanking, UTF-8)
   - select which debconf frontend is to be used (text, newt, gtk)
   - start main-menu (see next section)
   - halt or reboot the system (after main-menu exits)

Scripts in `/lib/debian-installer(-startup).d` can be architecture specific.

The Debian Installer Menu

The component main-menu drives the rest of the installation. It is responsible both for dynamically assembling the menu and for executing components when they are selected. Note that the menu is only actually visible when installing at low or medium debconf priority. At higher priorities it is still there, but it will automatically execute the next component without showing the menu to the user.

In some situations the debconf priority is automatically changed. It is reduced when the execution of a component fails, or when the user uses the <go back> button to back out all the way to the menu. In these cases it will also be increased back to the original level when the next component executed finishes successfully.

An important characteristic of udebs is that execution of their postinst scripts is delayed. On installation udebs are only unpacked; the execution of the postinst is done by main-menu and is actually what happens when a component is selected in the menu. In other words, main-menu can be said to be responsible for ”configuring” the udeb.

For a component to be included in the menu, it needs to have an Installer-Menu-Item line in the dpkg status file (`/var/lib/dpkg/status`). The order of components in the menu is determined primarily by its dependencies; the menu item number is used only where the order for two or more components cannot be resolved by dependencies alone.

Provides can be used to define virtual states which other components can depend on, as shown in the next example.

```
Package: netcfg
Status: install ok installed
Version: 1.23
Provides: configured-network
Depends: libc6 (>= 2.3.5-1), libdebconfclient0, libdebian-installer4 (>= 0.37),
         dhcp-client-udeb | dhcp3-client-udeb | pump-udeb, libiw28-udeb,
         cdebconf-udeb, ethernet-card-detection
Description: Configure the network
Installer-Menu-Item: 18

Package: choose-mirror
Status: install ok unpacked
Version: 1.19
Depends: libc6 (>= 2.3.5-1), libdebconfclient0, libdebian-installer4 (>= 0.38),
         configured-network
Description: Choose mirror to install from
Installer-Menu-Item: 23
```

This example also shows that netcfg has been run successfully ("installed") while mirror selection has not yet taken place ("unpacked").

Some components are included in the menu but are not run by default. These components have a menu number higher than prebaseconfig. Examples are components that allow to change the debconf priority, to save debug logs, to check the integrity of a CD-ROM or to abort the installation.

Hooks provide additional flexibility

At certain points in the installation, components provide run-parts like execution of scripts. This allows other components to just drop scripts there so that commands can be run at that point in the installation, even though the postinst for the component itself is run earlier or later (if the component even has a postinst).

The main hooks are:

/usr/lib/base-installer.d Run by base-installer before debootstrap is started.

/usr/lib/post-base-installer.d Run by base-installer just before kernel selection/installation.

/usr/lib/prebaseconfig.d Run by prebaseconfig.

Other, more special purpose hooks are `/usr/lib/apt-setup/generators`, `/lib/main-menu.d` and `/lib/rescue.d`.

Besides these general hooks, partman is basically structured as one big collection of hooks where partman components all drop their own scriptlets to extend partman's functionality. The most noteworthy of these hooks is `/lib/partman/finish.d` as some bootloaders drop scripts there to add checks that ensure the selected partitioning scheme conforms to their requirements.

Some special tools

The installer has some special commands which can be used in postinst or preseeding scripts:

anna-install Used to install additional, non-standard d-i components (udebs). It will check if anna has already been run. If it has, the component is unpacked immediately; if it has not, it will be scheduled for installation when anna is run.

apt-install Performs the same function for normal packages and installs them in the target system. Packages will either be installed immediately (if called after base-installer) or scheduled for installation as one of the "extra" packages installed near the end of base-installer's postinst script.

log-output Allows to run a command and redirect its output (either stdout or stderr or both) to `/var/log/syslog`.

in-target Allows to run a command in a chroot on `/target` with the chroot being set up for more demanding operations. For example, proc and sysfs are mounted and a policy-rc.d is created. Can obviously only be used after the base system has been set up.

Automating the install using preseeding

There are three preseed methods that are currently supported:

initrd preseeding The preseed file `/preseed.cfg` needs to be present in the initrd. It is read as part of the debian-installer-startup processing.

file preseeding Triggered by the presence of the *preseed/file* boot parameter.

network preseeding Triggered by the presence of the *preseed/url* boot parameter. In versions later than Etch Beta 2 it is also possible to trigger this from the DHCP server.

Which of these methods is available depends on the installation method.

The main purpose of preseeding is to set default values for debconf questions. Note that it makes no sense when using file or network preseeding to preseed values for questions that are asked before the preseed file is parsed.

Besides offering the option to set default values for debconf questions, preseeding also makes it possible to run scripts at two distinct moments using *preseed/early_command* and *preseed/late_command*. The *early_command* is executed immediately after the preseed file is parsed (only for file and network preseeding); the *late_command* is executed as part of the prebaseconfig component.

These scripts can be made as complex as you like. One option is to use them to install additional packages on the target system (using `apt-install`). The *early_command* could be used to install additional d-i components (using `anna-install`) or to install scripts in the various hook script directories (if commands need to be executed at a specific point in the installation).

Additional information about preseeding is available in an appendix of the Installation Guide.

Debugging the installer

Because most of the installer is scripted, it is fairly easy to debug most problems by adding a `set -x` in the correct place. The obvious place to start is the postinst of the component you want to debug. The output will appear in `/var/log/syslog`, which can most easily be viewed by starting the internal webserver from the "Save debug logs" menu option (after the network has been configured).

For components written in C debugging is a bit harder. Options are to use the strace udeb (add it to a custom image if needed) or to create a custom version of a component with added debug statements in the source and use that.

For debugging, you will probably want to control when components are started. Booting the installer with *install debconf/priority=medium* is a good way to achieve that. It will make sure the menu is displayed before a new component is started.

If you boot the installer at medium priority, you will also be able to load the optional component open-ssh-client[4]. Loading this component will allow you to use scp to copy files from the system being installed to another computer and vice versa. Also useful for testing new versions of scripts or commands without rebuilding the udeb and image. The command nc from the netcat package is available by default in the installer.

D-I components or udebs

A udeb (or micro-deb) is a special kind of Debian package. Technically udebs are very similar to regular packages: you can look at their contents using `dpkg -c`, and extract them using `dpkg -x` and `dpkg -e`.

The main difference is that a lot of policy requirements are waived. For example, a udeb does not contain a changelog, licence, manpages or md5sum[5]. The reason is to minimize size which is important as the installation completely takes place in RAM, with swap only becoming available after stage 4 of the installation (partitioning).

Another important difference is that udebs are not really meant to be uninstalled or upgraded.

The relaxed policy requirements make one of the reasons that udebs should to be installed on a normal system. The other reason being that it just doesn't make sense and it's likely to break things.

Two special classes of udebs should be mentioned here. However, covering these in detail is outside the scope of this paper.

Kernel image and kernel module udebs Kernel udebs are built basically by repackaging a regular kernel-image package. Reason is again to reduce memory usage: not all modules included in a kernel-image package are needed during an installation. Also, different modules are needed in the initrd for different installation methods, remaining

[4] If you want to load optional components when installing at the default priority, just back out to the menu and select the component that installs additional installer components.

[5] Of course, the source package for a udeb does need to contain a proper licence and changelog.

modules can either be loaded later or optionally (by manual selection or through dependencies). The package kernel-wedge contains the toolset used to reorganize a kernel-image package into multiple kernel (module) udebs.

Partman and its components Partman has a very specific structure and requires a fairly strict conformance to this structure for udebs that extend its functionality.

Contents of a udeb

For components that are included in the main menu, the udeb will at least contain:

- a postinst
- a debconf template that contains the description for the main menu:

```
debian-installer/<component>/title
Type: text
_Description: <menu entry>
```

Other things like additional debconf templates, C programs, hook scripts can be added as needed.

A special type of control file worth mentioning is the isinstallable file. If a script with this name is present in `/var/lib/dpkg/info` for a component, the main menu will run this script and only include the component in the menu if the script exits 0.

Creating a udeb

Creating a udeb is not all that hard, especially if you use debhelper. debhelper knows the special properties of a udeb and will do the right thing by default at build time. That is, if you don't forget to tell it you are creating a udeb.

The example below shows the `debian/control` file for a udeb that is supposed to be included in the main menu. Note the special section.

```
Source: kbd-chooser
Section: debian-installer
Priority: optional
Maintainer: Debian Install System Team <debian-boot@lists.debian.org>
Uploaders: [...]
Build-Depends: debhelper (>= 5.0.22), libdebian-installer4-dev (>= 0.41),
               po-debconf (>= 0.5.0), flex | flex-old , bison,
               libdebconfclient0-dev (>= 0.49)

Package: kbd-chooser
Architecture: i386 amd64 powerpc alpha hppa sparc [...]
XC-Package-Type: udeb
Depends: ${shlibs:Depends}, ${misc:Depends}, console-keymaps
Description: Detect a keyboard and select layout
XB-Installer-Menu-Item: 12
```

The line `XC-Package-Type` tells debhelper to treat the package as a udeb. The `XB-Installer-Menu-Item` is added in the control file for the udeb and will eventually end up in the dpkg status file to help main-menu figure out that this udeb should be included in the menu and in what order[6]. Packaging a udeb becomes a bit harder if it is derived from a regular package but needs to be compiled with different compiler options (e.g. some features disabled or a different optimization).

The main thing to always keep in mind when creating a udeb is size. It is very important to keep size as minimal as possible. This includes using tabs instead of spaces when indenting in scripts and not being too verbose in comments.

[6] The file `installer/doc/devel/menu-item-numbers.txt` in the d-i SVN repository documents menu numbers currently in use.

Library udebs

A major recent improvement is the addition of ”package type” support in shlibs files for libraries. This allows dpkg-dev and debhelper to automatically set correct dependencies on library udebs when a d-i component that depends on them is built.

For example, the regular binary package zlib1g now has the following lines in its shlibs control file:

```
libz 1 zlib1g (>= 1:1.2.1)
udeb: libz 1 zlib1g-udeb (>= 1:1.2.1)
```

The second line is specific to the package type ”udeb”. This alternative line is used when dpkg-shlibdeps is called with the `-tudeb` option; dh_shlibdeps will automatically add this option when processing a udeb.

Generating the extra udeb: lines is supported by dh_makeshlibs if the `--add-udeb="<udeb name>"` option is used. For example, the `debian/rules` file for libusb contains the following line:

```
dh_makeshlibs -V -s --add-udeb="libusb-0.1-udeb"
```

Building installer images

This paper will only provide an introduction to building installer images using existing definitions. The README in installer/build in the SVN repository contains more detailed information about the build system and how to modify existing or define new images.

An image consists of:

- a kernel;
- an initrd, which is basically a collection of unpacked udebs;
- in some cases a boot loader and/or configuration files used for booting.

Most d-i images are ”ready for use”. The exception are the cdrom images which form only the base (kernel and initrd) for creating the actual CD or DVD images. The package used for creating the CD/DVD images is debian-cd.

On some architectures there is one CD image that is ready for use: the mini.iso that is produced as a by-product of the netboot target. Reason is that this image does not really support installing from CD, it just allows booting from CD but retrieves all additional udebs and packages over the network.

It is important to distinguish between building images for release and building images for development/testing use.

A release build is done, as for other packages that are to be uploaded, from the installer directory using `debian/rules`. This will create a binary package (needed for uploading) containing some documentation, but the important bit is a tarball containing all installer images. After the upload this tarball needs BYHAND processing[7] by FTP-masters before the buildds will pick up the upload for other architectures.

Building images for development and testing is done from the installer/build directory[8] using `fakeroot make <target>`.

An important difference between release and development builds is that release builds will use udebs for the same suite as the target system being installed, while development builds will by default install testing, but use udebs from unstable[9]. This allows to mostly avoid the occasional breakage of the base system and tasks in unstable while using the most recent udebs.

Requirements for building

For both release and development builds the build dependencies as listed in `installer/debian/control` need to be satisfied.

To build installer images from SVN trunk, your build machine needs to be running unstable or you need to set up a sid chroot to build in. (To build images from the sarge branch of the repository, the build machine needs to run Sarge.)

During the build, the needed udebs will be retrieved from a mirror. By default this mirror is based on your `/etc/apt/sources.list` (see the generated file `build/sources.list.udeb`). To use a different source, create a file `sources.list.udeb.local`.

[7] This entails unpacking the tarball into the correct location on the master mirror server and creating/updating the correct symlinks. See for example http://ftp.debian.org/debian/dists/sid/main/installer-i386/.

[8] This includes the daily built images available from http://www.debian.org/devel/debian-installer. These are generated and uploaded from machines run by d-i porters using the daily-build script.

[9] This is accomplished by including the `/installer/build/unstable.cfg` preseed file in the initrd.

Build targets

To see which targets are available, run make. This will result in a list of some 130 targets, most of which are not really relevant. A more useful list can be obtained with `make | grep ^build`. The table below has the most often used targets for i386.

build_all	Builds all images
build_cdrom_isolinux	Builds the cdrom images (both 2.4 and 2.6)
build_netboot	Builds the netboot images (both 2.4 and 2.6) and the mini.iso
reallyclean	Completely cleans the build environment

The reallyclean target is often needed when changes are made between builds because otherwise udebs or information may be retrieved from temporary or cache directories and the changes will not take effect. The *rebuild_** targets clean some of this, but not always enough.

The build system explained

The easiest way to start is with the purpose of the subdirectories in the installer/build directory.

- `config`: defines the available targets (per architecture)
- `pkg-lists`: defines which udebs are included in an image (per image type)
- `boot`: contains configuration files and make targets used to make images bootable
- `localudebs`: allows to use (versions of) udebs not available on the mirror you use
- `util`: contains helper scripts called from the Makefile

Two files containing important configuration info are `config/dir` and `config/common`. However, normally there should be no need to modify any of the variables defined in these files.

Both the `config` and `pkg-lists` directories have a tree structure with general configuration defined in the root and more specific configuration defined in branches and leaves. Branches are defined in directories that have the same name as a config file on the higher level. The config directory contains makefile snippets.

config

For example, the definition for i386 images starts with `config/i386.cfg` which, besides the current kernel versions, defines the media supported with the line:

```
MEDIUM_SUPPORTED = cdrom netboot floppy hd-media
```

These media correspond to the main targets for i386 and are further defined in `config/i386`. The `netboot.cfg` file in that directory contains, amongst others, the following three lines:

```
FLAVOUR_SUPPORTED = "" 2.6
MEDIA_TYPE = netboot image
EXTRATARGETS = build_netboot_2.6
```

This defines that the netboot image has two flavors: the default one (using a 2.4 kernel) and an one using a 2.6 kernel, which is further defined in the `config/i386/netboot/2.6.cfg` file where the default values of the variables for the kernel version are overruled.

The files in config are processed recursively to dynamically generate the build targets, so in this example you get a netboot, a *netboot_2.6* target and targets for the other media. Building is also recursive, so calling the *netboot* target will automatically build both the netboot and netboot_2.6 images.

The structure of the config files can get quite complex and it can be hard to keep track of the exact role of the different variables set in them.

pkg-lists

The list of udebs to be included in an image is built by the `util/pkg-list` script based on definitions in the `pkg-lists` directory. Again, processing can be quite complex. Let's take the netboot target for i386 as an example to explain it.

First the file `pkg-lists/netboot/i386.cfg` is considered and all udebs listed in it are added. Some example lines from that file:

```
#include "discover"
console-keymaps-at
console-keymaps-usb
usb-discover [2.4]
socket-modules-${kernel:Version} ?
acpi-modules-${kernel:Version} [2.6]
```

The variable $kernel:Version is expanded to match the package name of the udeb based on the kernel version and flavor. If the name of a udebs is followed by "[2.4]" or "[2.6]", it is only included if the kernel major version for the image being built matches. If it is followed by a question mark it is skipped if the package is not available (without the question mark an error would be generated).

The first line with the `#include` results in the file `pkg-lists/discover` being processed next in the same way.

The `pkg-list` script will also always look for the presence of files named common and local and thus `pkg-lists/netboot/common` is processed next. This file exists and lists a number of udebs that belong in any netboot image, independent of the architecture. This file also contains two include directives:

```
#include "base"
#include "kernel"
```

Thus, udebs listed in `pkg-lists/base` (containing udebs common to all images) and `pkg-lists/kernel` (included in all bootable images) are also processed.

The file `pkg-lists/netboot/local` does not normally exist as it is intended for the inclusion of non-standard udebs. It is also very useful for testing as it can be used to temporarily add udebs not normally included without teh need to modify the regular files.

Finally, the script will check for `pkg-lists/local` and `pkg-lists/exclude`. The latter exists and contains some udebs otherwise pulled in by dependencies, but that should not be included because of library reduction, which is covered in the next section. Note that the exclusion if not triggered by the file name, but rather by the dash after the name of the udebs.

All dependencies of udebs listed in pkg-lists will also be automatically included in the image.

To see how the package list is built for a particular image, set `my $debug=1;` in the `util/pkg-list` script.

Result of the build

If the build is successful, you will find the images under the `build/dest` directory. Depending on the type of build you will also find manifest and log files there.

Before the image is created, its contents are assembled in the directory `build/tmp/<target>`. The tree subdirectory there contains the full contents of the initrd; other subdirectories are used for different purposes.

Library reduction

Library reduction (relinking a library leaving out unused symbols) is used as yet another method to minimize the size of initrds. The downside of library reduction is that this requires the dev and pic packages for the libraries to be reduced to be installed on the build system which also means that their version needs to match the version of the libraries in the udebs.

The size reduction is most significant for libc (40%) and libm (90%). Other libraries that are reduced include libresolv, libslang and libnewt. The reduction is done by calling mklibs from the main Makefile.

As only the executables that are included in an image are taken into account during the library reduction, we have provide for executables in components that are installed later as they would fail if they use symbols that have been taken out.

This is the reason that the udebs containing reduced libraries are excluded in `pkg-lists/exclude` which results in the udeb not being listed in the `/var/lib/dpkg/status` file in the intrd. If no udebs that are installed later

depend on the library, all is well. If a udeb that does depend on it is installed later, anna (or rather udpkg) will see that the dependency is not satisfied, and will install the udeb so the unreduced library replaces the reduced version.

Note that library reduction is only done after unpacking udebs for inclusion in an image; the libraries included in udebs are never reduced.

Using localudebs

The localudebs directory allows to use a different version of udebs than is available from the mirror you use. This can be used to test a new version of a udeb or to run the installer with a debug version of a udeb. It can also be used to build an image with a custom udeb.

To use a local udeb, just copy it into the directory. A Packages file will be generated automatically. Your udeb should have a version equal to or greater than the udeb currently on the mirror you use.

Note that local udebs will only be included in the image if the udeb would be included in a normal build too. So it has to be selected by the pkg-list script. Create a `pkg-lists/local` or `pkg-lists/<image>/local` to add udebs to the image that would not normally be included.

Some things to keep in mind when using localudebs.

- If you add an extra udeb, its dependencies will be included too. If those dependencies include virtual packages, the result is not always what you'd expect.
- Adding extra udebs will increase the size of the initrd; some architectures have limits for initrd size.
- If you use a `sources.list.udeb.local`, make sure to add as the first line:

  ```
  deb copy:<path-from-root-to>/installer/build/ localudebs/
  ```

- Don't forget to clean up after you're finished.

Conclusion

Hopefully this paper will help make Debian Installer more accessible to new developers. If you have any suggestions to improve this document, please mail them to the debian-boot list or the author. The intention of the author is to use this paper as the basis for a d-i developers reference.

For any kind of work on Debian Installer, you should check out the d-i SVN repository on alioth:

```
$ svn co svn+ssh://svn.debian.org/svn/d-i/trunk
```

Subscription to the debian-boot list is recommended. To request commit access to the repository, please send a mail to that list.

Some additional development oriented documentation can be found in the repository under installer/doc/devel or in README files included with the source for various components.

13. Let's Port Together. Debian Fun for Everyone

by Peter De Schrijver and Steve Langasek

Background : modern system architecture

Before diving into the details of portability, it's useful to have a quick look at the architecture of modern systems.

Figure 13.1 shows one typical system architecture. There is a CPU complex which interfaces using the frontside bus to the northbridge. The northbridge interfaces to the main memory subsystem, the PCI bus and the AGP. Most peripherals are directly or indirectly connected to the PCI bus. Exception is the graphics subsystem which has its own port the the northbridge, the AGP. This is due to the high bandwidth requirements of modern 3D programs. Another system architecture is shown in Figure 13.2. Here the main memory is connected directly to the CPU. The CPUs themselves are interconnected via Hypertransport, a relatively narrow but high speed interconnection technology. Peripherals are connected via PCI express. The PCI express lanes are connected to the system via a hypertransport PCI express bridge. PCI express is a link based interconnection technology using 1.2Gbit/s bidirectional lanes. Multiple lanes can be bundled together for increased bandwidth. In contrast to PCI, PCI express is not a bus. This offers advantages with regard to scalability, powermanagement and latency. It's interesting to see why modern interfaces move from parallel to serial. The main reason is performance. A parallel interface requires all the wires to be almost the same length. Put differently : the small differences in signal propagation times due to wirelength variations limit the clock speed of a parallel interface. However it became only recently possible to integrate the necessary electronics for serialization/deserialization on chips, which is why we didn't see this move earlier. We also saw a great increase in CPU clock frequency in the last years. Unfortunately I/O and memory interfaces did not follow this trend by the same amount. Bandwidth and latency did not improve by the same amount. This lead to the introduction of cache memories and burstmode transfers to hide the memory and I/O slowness.

C language portability pitfalls

The ANSI-C language standard defines a few rules with regard to types. Writing portable code means sticking to this rules and not assuming a particular size for a type, unless this is guaranteed by the standard. In general C programs should use int as much as possible for simple integer computations, loop variables etc. In case you need a specific size (to implement a file format or a binary communication protocol), the ISO C99 types are prefered. Some operations like basic arithmetic or shifts might be more expensive on these types though. Be aware of bitfields. They are particularly nasty because the ordering of the bitfields is architecture dependent. Newer gcc versions tend to catch more portability problems such as signed versus unsigned types etc. It's worthwhile to compile your code using a recent gcc version with the -Wall option to catch this sort of problems.

Endianess

Endianess refers to the ordering of bytes in a multibyte 'object' like a 16,32 or 64 bit integer. There are basically 2 ways : little and big endian. Using little endian ordering, 0x12345678 will be represented as 0x78 0x56 0x34 0x12. In big endian ordering this would be 0x12 0x34 0x56 0x78. This is important any time programs interchange binary data. Examples are network protocols, file formats, ... The best way of coping with this is to have proper 'serializing' and 'deserializing' routines which handle conversion of the data between the internal and the external representation.

Alignment

Alignment means a multibyte 'object' starts at an address which is a multiple of its size. Some RISC CPUs do not handle unaligned accesses in hardware (unlike IA32 for example which does handle an unaligned access transparently for software). Most RISC CPUs which do not handle unaligned accesses in hardware, will generate a trap when an unaligned access is detected. This trap is generally handled by the kernel, which will handle the access. This obviously

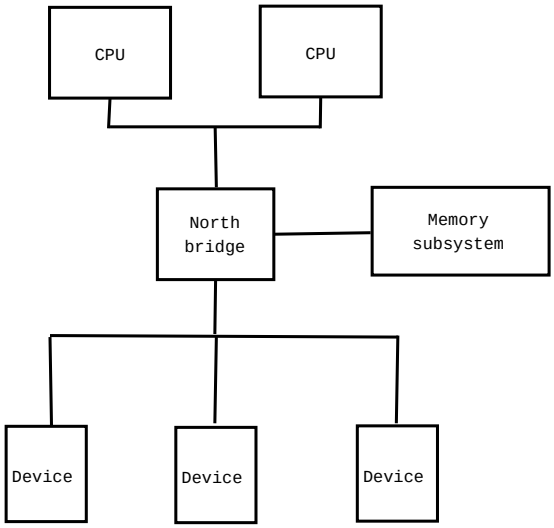

Figure 13.1.: Architecure of a modern system

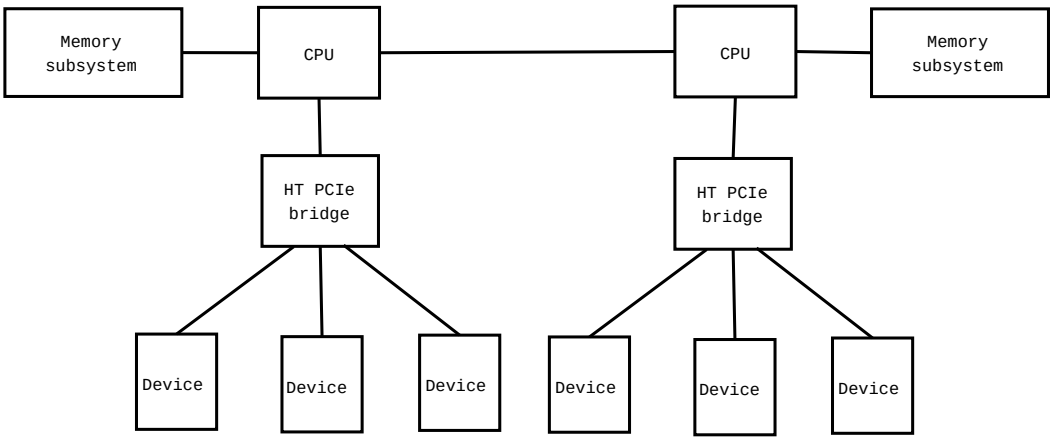

Figure 13.2.: Another system architecture

has significant performance impact. There are RISC CPUs which do not generate a trap, but basically ignore the lower address bits in case of an unaligned access. ARM is probably the most notable example of this kind of CPUs. Modern ARMs can actually generate a trap, but this feature is disabled for compatibility with older ARM CPUs. Unaligned accesses are almost never atomic, even on CPUs which handle them in hardware. This is important when implementing RCU or other look free data exchange mechanisms. In general it's best to avoid unaligned accesses as much as possible. In some cases the compiler can generate the right code for an unaligned access, but in many cases (particularly when using typecasts in C), it can not. In that case it might be necessary to copy the data to an aligned buffer first.

Memory organization

Memory organization refers to the way the userland address space is generally used by executables. A few things to be aware of : stack ordering and variable number of arguments in C. Not all architectures have a downward growing stack as IA32 has. Most notable is PARISC.

Accessing peripherals

Even though accessing peripherals is generally done by the kernel in a Debian Linux system, some userland programs also access peripherals. Most notably are probably the XFree86 and X.org X11 servers. Other examples include cdrecord and gphoto. There are generally 2 ways of accessing peripherals in a system. Either by directly writing to registers or memory of the peripheral, or by exchanging messages with the peripheral via eg USB, 1394, SCSI, ATAPI,... In the latter case the kernel generally cares about the transport of the data between the CPU / memory and the peripheral. This means most of the platform specific details are taken care of. Directly accessing a peripheral however is problematic with regard to portability. Inside the kernel most portability issues are dealt with by using special macros/functions to access peripheral registers. These macros/functions are however not available for userland programs. In general this means the userland program has to provide these abstractions itself.

14. Security Enhanced Virtual Machines – An Introduction and Recipe

Manoj Srivastava

Abstract

This paper, and the corresponding workshop, focusses on one of the major problem areas that any organization that is active on the Internet has to solve in order to conduct business in an increasingly hostile environment. The Discretionary Access Controls (DACs) that are the predominant Operating System (OS) techniques in mainstream OS's for managing security make them highly vulnerable to cyber-attacks, since they lack the ability to introduce and enforce strong, system-wide, security policy based, system defenses. This paper details the need for Mandatory Access Control (MAC), the benefits of virtualized server platforms and strong compartmentalization, and walks through a step by step process of implementing such a security architecture on a modern Debian system. This walk through would entail configuring and compiling an virtual machine (the example is an User Mode Linux [UML] image, but the same mechanism can be adopted for Xen VMs as well), creating a base root file system for the UML image to run, and briefly touches on the networking configuration required to connect the virtual machine to the network.

Introduction

In this increasingly connected information age, the utility of a computer is negligible unless it is connected to the internet. Also a truism in this age, that any machine connected to the Internet increasingly comes under attack. The trend is a growing spate of remote and local attacks on computer systems.

In face of this increasingly hostile environment, it is difficult for organizations to meet common security goals, including, but not restricted to:

Authentication This ensures that the actors initiating the actions taken by the system are correctly identified, and an assurance that the identity is not false.

Authorization This ensures that the actor is authorized to perform the action under consideration, or has access to resources on the system.

Confidentiality It ensures that the information on a computer system or transmitted over a network is only accessible to authorized entities. This applies to simply revealing the existence of the information, object, or asset.

Integrity It means information (or other assets) can only be modified or deleted by authorized entities, using authorized mechanisms, without bypassing auditing, for example. It is also required that there is some assurance that data has not been tampered with.

Domain Separation For information assurance, it is desirable to compartmentalize information into separate domain, with varying levels of secrecy and security. It is imperative that cross-domain information flow be properly scrubbed, and there be no mechanisms to bypass such scrubbing. Absent these cross-domain pipelines, leakage between domains must be prevented.

Availability It means that the assets are accessible to the authorized parties in a timely manner. Failure to meet this goal results in denial of service.

In this paper we concentrate on AUTHORIZATION, CONFIDENTIALITY, INTEGRITY and DOMAIN SEPARATION. We also touch on some mitigating stances that systems can take to partially counter denial of service attacks.

Motivation

Why do we need security and information assurance now more than we ever did before? Because there is an increasing frequency of active attacks based on software bugs, bypassing perimeter security using Trojans, man-in-the-middle

attacks, back doors, spy-ware, malware, distributed denial of service attacks and bot nets. Who are the attackers? Some are "script kiddies," while others are spammers and extortionists.

Another trend is for the attackers being increasingly motivated by the profit motif, and the attacks are no longer *hacks* with mischievous intent. Financial, and identity data, is increasingly coming under pressure, and we are only a short distance away from entities being engaged in a network warfare scenario for economic, political, and reasons of sheer spite. In this case, the situation is even more dire, with dedicated teams of opponents engaged specifically to destroy, disrupt, degrade, deny, delay, corrupt or usurp information resident in or transiting through mission critical networks. Have no doubt, this is indeed warfare, and most systems are vulnerable.

A typical scenario involves an attack vector that exploits software flaws in Internet facing services, and subsequently corrupts data residing in the service, or otherwise disrupts nominal operation. For example, suppose the target machine runs SSHD, which has a software flaw. The attackers send a carefully crafted, long string to SSHD, which fails to check length of input stream. The input buffer overflows into the stack. As a result, the attacker gets their code executed by SSHD, which runs privileged in most mainstream operating systems.The machine is now compromised, as is everything that trusts it. Once discovered, such a penetration costs many man-hours to correct. Attacker may, of course, be the authorized user of the machine, thus trying to get unauthorized privilege.

Some of the negative effects of these attacks can easily be mitigated by providing highly granular privilege separation in the service. However, very often this can not be done in the discretionary access control (DAC) model utilized by most current operating systems, since the service often owns the data objects whose integrity needs to be protected.

Another common attack vector results in cases where software flaws in system services are exploited remotely to gain privileges on the target platform often resulting in the attacker taking over the computer system in question. Such an escalation of privileges by the attacker can be hard to prevent in complex pieces of software executing on conventional COTS operating systems. Sand boxing applications and services and protecting them from each other and the underlying system would do a lot to mitigate such attacks.

Yet another family of attacks succeeds by getting an authorized user to run an infected program; and then this "Trojan" has access to any data available to the victim running it, as well as authorization to take any action that the user may legitimately take (like sending email). Given these privileges, the "Trojan" can corrupt data, send it back to the attacker, or erase it. In any case, data integrity and confidentiality are compromised. Lack of fine grained privilege separation leaves the victims open to such attacks. The damage from a Trojan is magnified manifold if it further exploits a local privilege escalation flaw (root exploit) and takes over the machine. In simple cases, the malware tries and infects other computers in the domain, and often these secondary attacks succeed since they are coming from a "trusted" computer in the local domain. In recent years, several worms have exploded over the Internet with wildly exponential infection rates by exploiting just such flawed software that was widely deployed.

In a more extreme but not uncommon scenario, a computer which is taken over may exhibit no immediate symptoms, but may wait for instructions from remote attackers to mount a coordinated attack (sometimes "phoning home" for instructions from the remote attacker) at a particularly critical moment.

The strategies employed by most operating systems to protect against such vulnerabilities (deploying anti-virus scanners and filtering firewalls) are reactive, and are ineffectual in the face of zero day exploits and the gap between an exploit being deployed and the security mechanism being upgraded to detect and disinfect the virus. Furthermore, rapidly mutating viruses present an even greater challenge to reactive, as opposed to pro-active techniques.

Complicating the situation even more in the distributed systems context, remote attacks, including distributed denial of service attacks, are devastating if not intercepted at the perimeter, and can bring application servers to their knees rapidly. Therefor the ability to distinguish and serve legitimate traffic becomes additionally critical for network defense.

In warfare (and modern business), information superiority has long been acknowledged as being critical to success. This implicitly dictates that we should be able to trust the integrity and provenance of the information that we have, and act upon. We must also ensure that the information we have is confidential, and if we operate in multiple security domains, with differing confidentiality requirements, that information flow from the secure to a less secure domain is appropriately scrubbed. This again implies that the information flow through the network must meet processing path guarantees, and must also meet security requirements for information flows, in a manner that can be audited and where we have some assurance that the security mechanisms and scrubbing procedures have not been bypassed.

Vulnerable programs characteristics

So what kind of programs are the targets of most of these attacks, and thus are high priority assets to defend? All these programs have some common characteristics. These characteristics include:

Privilege changes For example, any `setuid` or `setgid` executable. These programs normally have privileges not available to the user executing them. Any exploitation of weaknesses in the program code can lead to privilege escalation, and may compromise the machine.

Assumption of Atomicity This is specifically exploitable is there is a window between a security check, and performing actions based on that check – for example, checking access permission and opening a file, or deleting a symbolic link – which could have been re-targeted in the interim. In any case, this could lead to bypassing the security check by an attacker properly timing their attack vector.

Trusting the execution environment For example, programs that assume that they are loaded as compiled, or that their plug-ins are to be inherently trusted.

Trusting user input The sshd example above is a classic case of this kind of vulnerability. Programs need to be especially careful when dealing with user input, which could be maliciously formulated, or inadvertently not meet expectations. Even if no unknown users have access, the program is still vulnerable to insider attacks.

Executing mobile code This is becomming common with the growing popularity of interpreted languages, where code is squirted to a server over the network and executed.

Using shared resources This by itself is not a vulnerability, but a compromise of any program accessing shared resources may infect all other users.

Solutions

Given the attack vectors and exploitable vulnerabilities detailed above, what are the ways we can counter the threat? Any solution needs to provide as many of the following system characteristics as possible:

Privilege separation This helps contain any compromise of a subsystem from spreading, and moderates attack vectors that target programs performing privilege changes.

Fine grained access control This allows for tighter control of the security of the system without impacting ability of subsystems to perform the required tasks. This also prevents programs run by a user from compromising all the assets accessible to the user, if a proper security policy is in place. Properly deployed, this may obviate the need for privilege changes entirely for some programs.

Role Based access control This allows a single human to wear several hats, and lower the security vulnerability and privilege when working on tasks where it is not required.

Least Privilege No process of user should be given more privileges than they actually need. This can work hand in hand with the finer grained access controls and privilege sets lock down the security of a system. If a program or user does not have a privilege, then it can not be exploited to compromise the machine in the first place.

Limitation of error propagation A compromise of a single application server or subsystem should not lead to the compromise of the machine as a whole.

Resistance to privilege escalation This prevents some of the common exploits immediately – even if a program is compromised.

Assurance Confidence in the completeness and correctness of security policy in use

Protected Paths This provides secure communication guarantees of confidential and unmodified messaging across a mutually authenticated channel

Not be bypassed No security solution is worth anything if an attacker can just bypass the security mechanisms.

Most of these properties require operating system control of program execution, capabilities, and of all system information flow paths to minimize leakage of secrets, prevent insertion of malicious programs, and protect the integrity of system processes.

One of the strongest defenses possible is Security Enhanced Linux from the NSA, with its mandatory access controls, and policy based security (which is added to the discretionary access controls common to UNIX derivatives). It provides all the characteristics of a successful security mechanisms mentioned above. It has no concept of a “superuser”, and does not share the shortcomings of traditional UNIX security mechanisms (like depending on coarse grained access control and `setuid` or `setgid` binaries).

Properly written, a mandatory access policy can be used to set up a sandbox for any program (including any and all Internet facing services), such that they can not access resources and information unless expressly permitted. SELinux policies allow for sand-boxing applications to minimize the effect of a successful compromise.

A compromise of a single application server of subsystem may affect data integrity of that application, but does not pose a threat to other subsystems or the system as a whole. Using a static information flow analysis of the security policy, it is feasible to determine what domains would be affected by the compromise of any system, and steps can be taken to either tighten security policies, or to address recovery of the downstream domains in case of a compromise.

Why Mandatory Access Control?

In view of the failure of conventional discretionary access control mechanisms to provide attestable levels of security for computer systems and networks, mandatory access control (MAC), based on operating system level capabilities, is foundational. In order to provide system security, end systems need to be able to enforce the separation of information based on confidentiality and integrity requirements. This is not possible without active support from the operating system, since otherwise, any security mechanism built on top of operating systems lacking this ability can be bypassed. Without MAC, application security mechanisms are vulnerable to tampering and bypassing, and malicious or flawed applications can easily cause failures in system security. Without MAC, preferably leveraged upon hardware based trusted computing mechanisms, it is impossible to provide the needed data integrity, confidentiality, and domain separation with any level of assurance.

Standard discretionary access controls provided in most operating systems base access decisions on coarse grained user identity and ownership. To ensure a cohesive chain of trust, it must be possible to consider additional security-relevant criteria such as the role of the user, the function and trustworthiness of programs, and the sensitivity or integrity of the data. Under DAC, files are owned by a user and that user has full control over them, including the ability to grant access permissions to other users. The root account has full control over every file on the entire system. An attacker who penetrates an account can do anything with the files owned by that user – and if the user is `root`, has full control over the system. As a corrolary, there is no way to protect against a malicious super user.

Protection against malicious code is not possible using existing DAC mechanisms because every program executed by the user inherits all of the privileges associated with that user. Malicious programs are free to change the permissions associated with all of the user's objects, as well as disclose or alter the objects themselves. This problem is exacerbated by the fact that only two categories of users are supported, completely trusted administrators and completely untrusted ordinary users. Many system services and privileged programs must run with coarse-grained privileges that far exceed their requirements. A flaw in any one of these programs can be exploited to obtain complete system access.

As long as users have complete discretion over objects, it will not be possible to control data flows or enforce a system-wide security policy. Type enforcement, a critical capability provided by MAC enabled operating systems, allows static analysis and enforcement of data flow, facilitates privilege separation, guards against privilege escalation, and provides all the support mechanisms required to assure data integrity and confidentiality.

Policy based control is yet another critical feature provided by MAC enabled systems. When properly implemented, a detailed security policy enables a system to adequately defend itself and offers critical support for application security by protecting secured applications from being tampered with and being bypassed. It allows critical processing pipelines to be established and guaranteed. A fine grained, tightly configured security policy also enables strong separation of application privileges that permits the safe execution of untrustworthy applications in an isolated, secure sandbox. Its ability to limit the privileges associated with executing processes limits the damage that can result from the exploitation of vulnerabilities in applications and system services.

A MAC security policy, with well formed domain transitions, can also prevent privilege escalations from succeeding. Even a compromised application, overcome by, say, a buffer overflow exploit, would not allow the remote attacker to gain control of the machine, irrespective of whether the exploited application was running as a user process or some privileged system process. MAC enables information to be protected from legitimate users with limited authorization as well as from authorized users who have unwittingly executed malicious applications. Access to sensitive information can be tightly controlled, and tamper proof audit trails can be kept of all access to data. This enforcement of role based access control (RBAC) allows for flexibility without compromising security. The ability for systems to provide capabilities such as these is essential for the design and implementation of secure systems. For example, separating security officer and systems administrator roles makes attacks perpetrated by rogue insiders harder to carry out.

Deploying SELinux results in security policies that are amenable to static analysis, and information flow assurances that provide validation that there are no leaks from one security domain to another. Thus, in the event of a breach in security, the scope of the breach can be readily assessed, and damage limiting mechanisms can be deployed.

Expanding on the above capabilities, SELinux also adds network path protection, where the concept of firewalls is extended to processes on a MAC enabled node. Network access policies will allow a process in a certain security context on one machine to be assured of a connection from another process in a known security context on a remote machine, as opposed to blocking packets based on IP addresses and port tuples as conventional firewall based approaches do. Even with asymmetric security (with MAC based labeling only on one end of the connection), a MAC enabled system can enforce policies based on roles and process domain policies, as opposed to the crude host-to-host policies of a firewall based solution. Once network paths and the packets traveling over them are labeled, it is possible to distinguish legitimate

traffic from potentially dangerous traffic, and block it before it reaches application code. This also helps mitigate denial of service attacks, by blocking unauthorized traffic at the perimeter, and not exposing services to such attack vectors.

The role-based access control component defines an extensible set of roles. Each process has an associated role. This ensures that system processes and those used for system administration can be separated from those of ordinary users. The configuration files specify the set of domains that may be entered by each role. Each user role has an initial domain that is associated with the user's login shell. As users execute programs, transitions to other domains may, according to the policy configuration, automatically occur to support changes in privilege.

Non-by-passable processing pipelines

The need to transfer data between security domains over a network is a common requirement today. Firewalls, email gateways, and other edge-of-the-network protection devices are some examples of such cross domain devices. A cross-domain solution (also called guards) that connects different security domains with differing levels of confidentiality and integrity requirements needs a high degree of confidence in its implementation.

A critical requirement of such a solution is that the processing pipeline of filters and scrubbers not be by-passable, and that data be transmitted across the device in a controlled way.

SELinux and MAC policies allow us to define a data flow path which can only happen through the required processing stages.

Eamples of functionality enabled by MAC

With MAC, one can ensure that a mail user agent run by an user only has access to files related to stored mail – and not all files owned by that user. MAC in effect provides each application with a virtual sandbox that only allows the application to perform the tasks it is designed for and explicitly allowed in the security policy to perform. For example, the webserver process may only be able to read web published files and serve them on a specified network port. An attacker penetrating it will not be able to perform any activities not expressly permitted to the process by the security policy, even if the process is running as the root user. Files are assigned a security context that determines what specific processes can do with them, and the allowable actions are much more finely grained than the standard Unix read/write/execute controls. For example, a web served file would have a context allowing the apache process to read it but not execute or make changes to it, while the log files would be appendable but not readable or otherwise changeable by apache. Network ports are also assigned a context, which can prevent penetrated applications from using ports not permitted to them by security policy. Standard Unix permissions are still present on the system, and will be consulted before the SELinux policy when access attempts are made. If the standard permissions would deny access, access is simply denied and SELinux is not consulted at all. If the standard file permissions would allow access, the SELinux policy is consulted and access is either allowed or denied based on the security contexts of the source process and the targeted object.

In conclusion

Role based access controls, type enforcement, auditable, enforced security policies, strong support for security domains, and a grounds up security policy designed around the principles of privilege separation and least privilege, can move any network operation into a highly defensible security stance. Such a system is pro-active, fail safe, proof against zero-day exploits of program flaws, provides strong separation of security domains, ensures application, and data integrity, ability to limit program privileges, can provide processing pipeline guarantees allows applications to be effectively sand boxed so that a compromised application does not affect other applications and services on the system, and provides an ability to strongly protect audit logs from unauthorized access and from tampering. It can implement authorization limits for legitimate users.

Further, it is undergoing common criteria security evalutaion at various levels, sponsored by various commercial distributions of Linux, so the security offered by SELinux and the reference policies has been certified by domain experts.

The contrast between this approach and the approach of most security products in the anti-virus and intrusion prevention and detection markets could not be more stark. Anti-virus and IDS/IPS systems based on signatures are reactive, operating only on known threats, which is why zero-day exploits are so prized by malware authors. You can compare these products to firewalls with a default "allow any" rule, and many specific "deny" rules. This is a losing battle, as the quantity of malware keeps increasing at an exponential rate and vendors and their customers fight a losing battle to keep up. Any newly discovered security flaw will have a window of vulnerability between the exploit's release and the signature being added and propagated to the end user.

Why Virtualization?

Virtualization offers security benefits in its own right; it provides strong data isolation, and can be used to setup multiple services on the same hardware in strict isolation from each other, which helps contain infection. So a compromise of a service on one virtual machine can be quarantined, and the virtual machine can be rebooted from known good data without affecting other services running in other virtual machines.

The best defense against external threats is not to let them in in the first place. The physical separation, or "air gap" defense, has become standard in high assurance computing environments, where separate networks are disconnected from each other. Thus, users needing physical access to multiple security domains must employ a separate CPU and monitor for each domain.

Virtual machines allow the administrator to easily set up multiple security zones on the same hardware, with total domain isolation between all virtual machines and the host machine, and implement different security policies as appropriate for the security zone (think if DMZ and internal servers on the same physical box).

In situations where attestable data separation is important, the ability to show data cannot flow between networks or between one VM to another is an important property – and can be achieved if the host machine also implements mandatory access controls.

Additionally, in conjunction with strong MAC security policies, it allows the administrator to finely tailor security policies for each specific virtual machine, and the services that machine is running.

Virtual machines with copy-on-write virtual file systems can be rolled back to a known good state fairly easily, which is always good if a particular application server was exploited.

Need for small severs and migration

Given the advantages of mandatory access control, and the serious flaws that it mitigates, why has the solution not become more popular and implemented in mainstream operating systems? MAC has been implemented in research operating systems for the best part of a decade, with clear and significant benefits. The stumbling block is that while the security advantages are impressive, the system is notoriously hard to configure, the major obstacle being in the difficulty in implementing a coherent security policy. Though current implementation of modular policy modules promises to reduce complexity and make it easier to incrementally evolve security policy, it is still a high skill task with a steep learning curve to develop policy from scratch. Setting up SELinux is not a task for the faint of heart, and the security polices currently extant are far from complete, making it almost impossible for most folks to convert a working machine to a secure box, and raises the bar for people who just want to casually try out SELinux. Anything to automate this process would help in increasing the security all around. This paper sets out to address these deficiencies.

One possible solution is to utilize virtualization, and instead of trying to convert a full featured, working desktop into a secure platform (quite hard, in advance of Security Enhanced X), and instead create a User Mode Linux virtual server running in strict mode. One of the advantages of running a UML is that we can create a read only root file system, and use copy on write file systems to ensure that any changes can be quickly reverted, even if someone can discover a flaw in the security policy, and exploit it. Also, with UML's, the monitoring mechanisms are out of the ken of the virtual machine, since they can run on the host machine, making it far harder to suborn them.

Methodology

In this paper, I am concentrating on a walk through for a UML virtual machine. However, most of the steps taken can be adapted for Xen, with minor changes.

There are a number of problems that novice users face with trying to use a virtual UML instance, firstly, the user-mode-linux package in Debian is showing signs of neglect, and, secondly, is not generally patched to support SELinux. Then there is the issue determining a compatible set of sources, patches, and sources of the patches (though as more and more patches get accepted into the mainstream kernels this is less of a problem now than it used to be).

Even when one has a proper /usr/bin/linux binary, there is the issue of finding a proper root file system to run the UML on. The root file system creation tools in Sid also show signs of neglect, and even then, one would need to install SELinux on these root file systems, which is often a frightening task by itself.

Compiling the host system

There is little to be done here, except to apply the SKAS patches to the host kernel. These patches allow the UML kernel to run in an entirely different host address space[1] from its processes. This solves the security and honey pot fingerprinting problems by making the UML kernel totally inaccessible to UML processes. Their address spaces are identical to what

[1] `http://user-mode-linux.sourceforge.net/skas.html`

they would be on the host. A given version of UML guest will look for a specific SKAS patch in the host and fall back to thread tracing mode if it's not there. The boot-up message will tell you which version it's looking for in case you're not sure. The steps are as follows:

1. Download the original kernel sources from kernel.org - selecting the latest version that seems to work well with UML, which is, at the time of writing, 2.6.16.1.

```
% cd /usr/local/src/kernel
% wget ftp://ftp.us.kernel.org/pub/linux/kernel/v2.6/linux-2.6.16.1.tar.bz2
% wget ftp://ftp.us.kernel.org/pub/linux/kernel/v2.6/linux-2.6.16.1.tar.bz2.sign
% gpg --verify linux-2.6.16.1.tar.bz2.sign linux-2.6.16.1.tar.bz2
```

2. Untar the sources somewhere (`/usr/local/src/kernel/` is what I use). It is really immaterial where the kernel is unpacked, but I tend to avoid /usr/src since I do not want to be root when compiling the kernels, and so the working directory I use has write permissions for an unprivileged user.

```
% tar jvvfx linux-2.6.16.1.tar.bz2
```

3. Create a dir for compiling the host kernel (`2.6.16.1, for example`). It is nice to compile the kernel in a separate directory tree from the place it has been unpacked, using a symbolic link farm, since it allows one to apply patches to the build tree while retaining the pristine source tree. Once can then use incremental patches when the next upstream release of the kernel comes out. Additionally, this allows us to build a host and a guest kernel (which needs different patch sets) without having to duplicate all the kernel source tree .

```
% cp -lr linux-2.6.16.1 2.6.16.1
```

4. Get the SKAS host patch. You can find it on the download page[2] for user mode linux. It is also a good idea to check out the download page of the author[3], which has updates. At the time of writing, I got the skas-2.6.16-v9-pre9.patch.bz2 [4] file.

```
% cd ..
% wget http://www.user-mode-linux.org/[...]/skas-2.6.16-v9-pre9.patch.bz2
```

5. Apply the SKAS patch.

```
% cd 2.6.16.1
% bzcat ../skas-2.6.12-rc4-v9-pre4.patch.bz2 | patch -p1 dry-run
% bzcat ../skas-2.6.12-rc4-v9-pre4.patch.bz2 | patch -p1
```

6. Configure the host kernel (without ARCH=um, this is a host kernel), enable `/proc/mm` under "Processor type and features" menu if needed, **save the new configuration** and build it.

```
 % make xconfig
% make-kpkg --rootcmd fakeroot kernel-image
```

7. Install the resulting package, tweak your boot loader as needed, and you are good to go.

```
 % dpkg -i../kernel-image-2.6.16.1-skas3-v9-pre9_2.6.16.1_i386.deb
```

If we were trying to compile a Xen virtual machine here, we would be compiling the domain 0 kernel image, which would mean a slightly different `.config` file, and not bothering with the SKAS patch, but not very different.

A recipe for compiling the UML image

The good news is that the uml patche is now incorporated into the mainline kernel. The mainline SELinux support is also there, for the most part, with occasional patches once in a while. The SELinux kernel patch for 2.6.16 includes a few minor changes pending merge in the next kernel release.

1. First, get the latest SELinux patches from NSA's download page[5].

[2] http://user-mode-linux.sourceforge.net/dl-sf.html
[3] http://www.user-mode-linux.org/~blaisorblade/
[4] http://www.user-mode-linux.org/~blaisorblade/patches/skas3-2.6/skas-2.6.16-v9-pre9/skas-2.6.16-v9-pre9.patch.bz2
[5] http://www.nsa.gov/selinux/code/download5.cfm

```
% wget http://www.nsa.gov/selinux/patches/2.6.16-rc6-selinux1.patch.gz
```

2. Create a dir for compiling the UML kernel (`uml-2.6.16.1`, for example), and populate it with symbolic links.

```
% cp -la linux-2.6.16.1 uml-2.6.16.1
```

3. Apply the SELinux patch. Note that there shall be a small failure, for Makefile, since the SELinux patch was against 2.6.16-rc6, and we are actually using 2.6.16.1. This is harmless.

```
% zcat ../2.6.12-selinux1.patch.gz | patch -p1 --dry-run
% zcat ../2.6.12-selinux1.patch.gz | patch -p1
```

4. Please note that if you configure something as a module, an extra step would be required to install the modules into the UML

   Configure the kernel (don't forget ARCH=um). Since we have patched the host kernel with SKAS, we may turn of the thread tracing mode. Also, hostfs is a nice option to have, unless you are very concerned about security and leaking information from the host system into the UML. Configure the character, block, and network devices to include all the features you shall need in the UML. I strongly suggest using the honey-pot proc pseudo file-system, as well as logging. **DO NOT** turn on SMP and `highmem` support; that is known to be broken for these versions. (Oh, don't forget to turn on SELinux – which also needs AUDIT to be turned on, amongst other things). **Save the new configuration** and build it.

```
% make ARCH=um xconfig
```

5. Recent versions of *kernel-package* have the functionality to build linux-uml packages natively, so, instead of doing `make ARCH=um linux`, we can build a linux-uml debian package instead.

```
% make-kpkg --arch=um --rootcmd=fakeroot kernel-image
```

6. This results in a `../kernel-uml-2.6.16.1-selinux1_10Custom_i386.deb`, for example, which can be installed anywhere using `dpkg`.

```
% dpkg -i ../kernel-uml-2.6.16.1-selinux1_10Custom_i386.deb
```

Preparing to network the UML's

The networking page[6] at the UML home defines a number of ways to network the resulting UML's. Since the kernels used are way beyond 2.2.X, ethertap was out. Multicast, slip, slirp, and pcap were also out – one should be able to use the UML to provide real world services. That leaves TUN/TAP and the Daemon protocols. The root_fs created below primarily supports the Daemon (since that makes the root_fs more portable, however, tuntap can also be used by just editing `/etc/network/interfaces` on the root_fs and uncommenting the tuntap stanza.

Using TUN/TAP

If you go the tun/tap route, you may either setup a fixed tap device as detailed below, or you may use the `uml_net` command. To do that, make sure that the user that runs the UML is in the group `uml-net`.

```
# adduser srivasta uml-net
```

Next, make sure `/usr/lib/uml` is in the path for the user who runs the UML

```
# export PATH=''$PATH:/usr/lib/uml''
```

After this, you just need to specify that the `eth0` interface in the UML needs to use the tun/tap transport, set up `/etc/network/interfaces` inside the UML, and then sit back and let uml_net do the rest (including proxy arp and all). An example is Appendix 14 on page 88.

```
# eth0=tuntap,,,<IP of Host Machine>
```

[6] `http://user-mode-linux.sourceforge.net/networking.html`

Using the Daemon transport

If you just want a bunch of UML's to talk to each other, then you are all set – `uml_switch` comes set up out of the box for you. However, if you choose to have the UML communicate to the external world, you need to set up a tap device for the uml_switch network to talk to. All you have to do is add something like this to `/etc/network/interfaces` (I chose to use tap2 since I sometimes use a vpnc client that likes to take up tap0; and *192.168.3.X* is close enough to my internal network that I would have to make minimal changes to firewall rules).

```
 iface tap2 inet static
  address 192.168.3.2
  netmask 255.255.255.0
  tunctl_user uml-net
```

Next, make sure that the tap2 interface comes up, tell uml_switch to listen to tap2, and restart uml_switch.

```
 # ifup tap2
# perl -pli.bak -e \
> 's/^#\s*UML_SWITCH_OPTIONS=.*/UML_SWITCH_OPTIONS=\``-tap tap2\''/'
> /etc/default/uml-utilities
# /etc/init.d/uml-utilities restart
```

Again, you may set up a static network interface inside the UML, or you can set up a DHCP server on the host, and setup your UML to be a DHCP client. The advantage of the latter is that I can then share root file systems across machines, since there is nothing that is site-specific inside the root_fs. An example of the static interface setup is in Appendix 14 on page 88.

```
 auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
  address 192.168.1.13
  netmask 255.255.255.0
  network 192.168.1.0
  broadcast 255.255.255.255
  gateway 192.168.1.10
```

To set up DHCP, first one needs to setup DHCP; Appendix 14 on page 89 contains an example you may use to set up dhcp on the host. You may need to edit `/etc/default/dhcp` in order to tell dhcp to listen on tap2, by adding the following line (if you already have an interfaces line, add tap2 to it).

```
 INTERFACES=``tap2''
```

Then just restart dhcpd to have it start listening on the tap2 line:

```
 % /etc/init.d/dhcp restart
```

Then mount the UML root_fs, and change the `/etc/network/interfaces` to the following, and you should be more or less good to go.(In my case, I also had to add tap2 to `/etc/shorewall/interfaces`, and also add tap2 to `/etc/shorewall/masq`, as well as allowing the IP address 192.168.3.X to access my local bind daemon).

```
 auto lo
iface lo inet loopback

auto eth0
iface eth0 inet dhcp
```

At this point, the possibilited expand. You can set up VPN's inside the UML instances, or use `brctl` and create a bridege to a virtual network of UML instances. You can post forward ports on the host system to ports on the virtual machine, and run all network facinf deamons inside the UML.

Creating a root file system

The first thing we need to be running a UML on a machine is to install the `uml-utilities package`.

```
 # aptitude install uml-utilities
```

Though there are already mechanisms in Debian for creating a user mode linux root_fs system (notably, rootstrap[7] they did not work for me. For example, rootstrap fails currently on a 2.6.x host, since rootstrap tries to build the root_fs inside a UML that uses hostfs to mount the current / as the initial file system - and proceeds to fall flat on its face, since libc assumes that it can use the native posix thread library (since the UML kernel version is 2.6.16.1), and `/lib/tls` exists – but UML has not yet ported NPTL over, so things fail.

I decided to write a simple shell script based around debootstrap[8], which also happens to be a simple shell script with very few dependencies, and hence fewer things that can go wrong. This shell script sets up a root_fs, based on a few variables at the top (you may also drop these variables in `/.creatfsrc`), and sets it up with a simple uml_net based networking.

First, select a dir on a file system that has at least a GB of space available, and run the the creatfs.sh script (after suitable modification).

```
 % cd /scratch/sandbox
% /path/to/creatfs.sh
```

At this point, you should have a root file system that can bootup, and while it is not yet running SELinux, it is ready to be taken there. There is a script written into `/root/post-install.sh` that needs to be run inside the UML to complete the process.

If you had configured anything in the UML as a module, this is the time to install them. If not, you may skip this step.

```
 % cd /scratch/sandbox
% mount -o loop root_fs mounted
% cd /usr/local/src/kernel/uml-2.6.16.1
% make INSTALL_MOD_PATH=/scratch/sandbox/mounted \
>  ARCH=um modules_install
% umount /scratch/sandbox/mounted
```

Russel Coker has created a very useful site[9] that has tweaks required to make a system work properly with SELinux; creatfs.sh tries very hard to incorporate all the relevant tweaks in the root_fs created. You should be able to fire up your UML like so if you are using the TUN/TAP interface and not the persistent tap2 device (just tweak the IP address of the host):

```
 % /usr/bin/linux mem=256M \
>  con=xterm con0=fd:0,fd:1 con1=xterm devfs=nomount \
>  tty_log_fd=3 3>tty_log_file \
>  eth0=tuntap,,,192.168.1.10
```

If, on the other hand, you are using the daemon transport, use:

```
 % /usr/bin/linux mem=256M \
>  con=xterm con0=fd:0,fd:1 con1=xterm devfs=nomount \
>  tty_log_fd=3 3>tty_log_file \
>  eth0=daemon,,unix,/var/run/uml-utilities/uml_switch.ctl
```

The root file system created in this step would also work with a Xen virtual machine, using a loopback mount. Networking with the Xen instance is different in detail, but conceptually identical.

Working on SELinux

As mentioned before, please look at the tweaks[10] page and make any changes that `creatfs.sh` may have missed. The next step would be to fire up the UML, and install the SELinux packages. To finish the process, do the following:

1. Fire up the UML. This shall be running in permissive mode, and as yet, there is no policy installed. Expect to see a lot of warnings in this run; but after we reboot the UML, it should be running with no warnings. So, as before, if using TUN/TAP transports, use:

```
 % /usr/bin/linux mem=256M \
>  con=xterm con0=fd:0,fd:1 con1=xterm devfs=nomount \
>  tty_log_fd=3 3>tty_log_file \
>  eth0=tuntap,,,192.168.1.10
```

[7] http://packages.debian.org/rootstrap
[8] http://packages.debian.org/debootstrap
[9] http://www.coker.com.au/selinux/tweaks.html
[10] http://www.coker.com.au/selinux/tweaks.html

Or else, use:

```
 % /usr/bin/linux mem=256M \
>  con=xterm con0=fd:0,fd:1 con1=xterm devfs=nomount \
>  tty_log_fd=3 3>tty_log_file \
>  eth0=daemon,,unix,/var/run/uml-utilities/uml_switch.ctl
```

2. Next, login as root, and run the final step inside the UML. Please note that installing selinux-policy-default fails initially, and generates error messages, but the script should clean all that up at the end.

```
 % /bin/bash /root/post-install.sh
```

3. Now, halt the UML

```
 % shutdown -h now
```

4. Fire up the UML a last time. This time around, there should be no warnings as you boot into an SELinux enabled UML. Enjoy.

```
 % /usr/bin/linux mem=256M \
>  con=xterm con0=fd:0,fd:1 con1=xterm devfs=nomount \
>  tty_log_fd=3 3>tty_log_file \
>  eth0=tuntap,,,192.168.1.10
```

Or else, use:

```
 % /usr/bin/linux mem=256M \
>  con=xterm con0=fd:0,fd:1 con1=xterm devfs=nomount \
>  tty_log_fd=3 3>tty_log_file \
>  eth0=daemon,,unix,/var/run/uml-utilities/uml_switch.ctl
```

Conclusion

At this point, you should have a mostly working SELinux virtual machine, barring major bugs in the SELinux packages. It should be easy to byuld up upon the `createfs.sh` script to create specialized root file systems, for isntance, creating a root file system that additionally has postfix installed, or the bind daemon, and mostly automate the creation of small, tightly controlled, server applications.

This is a work in progress, I'll be updating the recipe on the web site `http://www.golden-gryphon.com/software/security/selinux-uml.xhtml` to work with Xen virtual machines.

The area which needs the msot attention at the moment is the referencxe SELinux policy, it has to be tuned for Debian, and we need modules to cater to the huge number of packages that we have but Fedora does not.

Static network interface

```
auto lo
iface lo inet loopback

# The first network card
# This entry was created during the Debian installation
auto eth0
iface eth0 inet static
    address 192.168.1.13
    netmask 255.255.255.0
    network 192.168.1.0
    broadcast 255.255.255.255
    gateway 192.168.1.10
```

DHCP configuration file

```
# dhcpd.conf
#
# Sample configuration file for ISC dhcpd
#

# option definitions common to all supported networks...
option domain-name "internal.golden-gryphon.com";
option domain-name-servers glaurung.internal.golden-gryphon.com;
option time-servers 132.236.56.250, 130.203.1.10, 198.82.162.213;

option subnet-mask 255.255.255.0;
default-lease-time 6000;
max-lease-time 72000;

subnet 192.168.1.0 netmask 255.255.255.0 {
  # range dynamic-bootp 204.254.239.33 204.254.239.40;
  range 192.168.1.100 192.168.1.120;
  option broadcast-address 192.168.1.254;
  option subnet-mask 255.255.255.0;
  option routers tiamat.internal.golden-gryphon.com;
}

subnet 192.168.3.0 netmask 255.255.255.0 {
  range 192.168.3.10 192.168.3.63;
  option broadcast-address 192.168.3.254;
  option subnet-mask 255.255.255.0;
  option routers 192.168.3.2;
}

# Fixed IP addresses can also be specified for hosts.   These addresses
# should not also be listed as being available for dynamic assignment.
# Hosts for which fixed IP addresses have been specified can boot using
# BOOTP or DHCP.   Hosts for which no fixed address is specified can only
# be booted with DHCP, unless there is an address range on the subnet
# to which a BOOTP client is connected which has the dynamic-bootp flag
# set.

group {
  use-host-decl-names on;

  host cinder {
    option ip-forwarding on;
    hardware ethernet 00:01:02:9C:DB:8E;
    fixed-address cinder.internal.golden-gryphon.com;
  }

  host ember {
    hardware ethernet 00:10:A4:E3:F1:9F;
    fixed-address ember.internal.golden-gryphon.com;
    option routers tiamat.internal.golden-gryphon.com;
  }
}
```

Part III.

Round tables

15. State of the Art for Debian i18n/l10n

by Christian Perrier and Javier Fernandez-Sanguino

Abstract

Debian GNU/Linux is one of the most ambitious free software project, putting together a large number of developers from all around the world working in building a free operating system.

This document describes the rationale and management of internationalisation and localisation within the Debian project, including the current infrastructure and tools available to both translators and package maintainers.

Keywords: free software, translation, internationalisation, localisation, operating system

Introduction

The following document describes the approach taken for handling internationalisation and localisation in the Debian project describing the infrastructure, and tools available for translators and package maintainers as well as the work done by the different translation team members.

English as the official language?

English is often regarded as the official language of most free software projects. This assumption affects users of free software and users of information systems generally, to the degree that "They should all just speak English!" is a frequent reaction to internationalisation projects. On behalf of the majority of the world's population - who do not speak English - a great deal of work is done voluntarily in the Debian project, to ensure that program interfaces do not only present their messages in English and that most documentation is not only written in English.

Currently, most of the working groups within Debian use English as their main language for information exchange (discussions, documentation, etc.). Developers, even if not native speakers, program code with an English interface and write documentation in English. The main information sources of official documentation (including the project website) are first written in English.

A single working language helps coordinate work of multiple developers worldwide, and helps people exchange their ideas. English is the common language for communication with Debian developers[1]. However, using only English disadvantages many potential users and limits the pool of potential developers. It's an accessibility issue, as well as a usability issue. We want everyone to be able to use their computer effectively, regardless of the languages they are familiar with or can understand, and that's what internationalisation is about.

The importance of internationalisation and localisation

Debian's goal of providing a *universal* operating system is based on the voluntary work of many translation teams. Having a *universal* operating system does not only mean having all possible software tools available, it also means providing an operating system that can be used by anyone worldwide. Even though English is a very widely-used language in the IT world today, this does not mean that all the world population can speak it, or even understand it. As we have pointed out, in fact, the majority of the world's population cannot use English. That is why this goal is so important and why we need to fulfill it, at least partly, by providing an operating system accessible to every user, in a language they are familiar with, regardless of what that language is. The more languages we support, the more people we reach. It's as simple as that.

This means Debian needs to provide an installation system that can also be used by non-English speakers, and that can set up a system also suitable for use for non-English speakers. Debian users need to be able to read about the operating system itself, itself, so documentation must be translated, as well as application interfaces. Both the documentation available in the operating system itself, and the documentation available through other sources (e.g. the project's website). Debian users need to be able to ask for help, so mailing lists, IRC channels, fora and other interactive help systems need be available to them in a language they can use.

[1] `http://www.debian.org/contact.en.html`

Making Debian usable by more people is also a selfish goal for the project: having more users means that the software developed by the project gets used more (more exposure leads to more testing and eventually, more features) and it also means that the new users might eventually contribute to the project (through bug reports, patches, translations) and even become Debian developers themselves. More access simply means more resources.

The internationalisation and localisation process

According to Introduction to i18n[2] from Tomohiro KUBOTA: 'I18N (internationalisation) means modification of a software or related technologies so that a software can potentially handle multiple languages, customs, and so on in the world.' while 'L10N (localisation) means implementation of a specific language for an already internationalised software.'

Internationalisation is the process that makes a program capable of providing a computing environment for the user adapted to his/her own language, currency, date and time formats, etc. Most users of a similar background will share an environment, and this means there a number of common environments that a program has to support. The term internationalisation is often abbreviated to i18n.

Even if a piece of software is ready to use different environments, that does not mean that it can immediately do so, since specialists (typically people who use that environment) need to adapt it so that it can be used in a language a user is familiar with. This process of adapting the software to a specific environment is called localisation. Localisation is typically focused on the translation of each and every message that the program can handle (and implements in the user interface, such as menus, buttons, error messages and tooltips). Localisation is typically abbreviated to l10n.

L10n and I18n are closely related, but the issues in achieving each of them are very different. It's not really difficult to allow the program to change the language in which texts are displayed, based on user settings, but it is very time consuming indeed, actually to translate all its messages. On the other hand, setting the character encoding may be trivial, but adapting the code to use several character encodings can be a challenging and complex task.

This document will not try to explain all the different issues facing localisation and associated with the representation of different code pages. However, it's important to recognize that this is a key issue, especially for environments that do not use the Western code pages (ASCII character set and ISO-8859), such as Asian and Indic scripts and languages with combined diacritics. We need to bear in mind that the majority of potential users world-wide use languages which are not covered by those two very limited code pages. UTF-8 support is a crucial step in internationalization.

Internationalising, translating and being internationalised and translated

Debian supports an ever-increasing number of languages. Even if you are a native English speaker and do not speak any other language, it is part of your duty as a maintainer to be aware of issues of internationalisation. Internationalisation is the bridge between you and most of your potential users. Therefore, even if you yourself can function effectively in English, this chapter gives you information you need as a maintainer.

Like the overall i18n implementation, which still has to coalesce into a standard process, l10n within Debian does not yet have a central infrastructure which could be compared to the dbuild mechanism for porting. Currently, most of the work has to be done manually.

How are translations managed within Debian? Handling translation of the texts contained in a package is still a manual task, and the process depends on the kind of text you want to see translated.

For program messages, the gettext infrastructure is in common usage. For applications the translation is handled upstream, within projects like the Translation Project[3], the Gnome Translation Project[4] or the KDE Translation Project[5]. The only centralised l10n resource within Debian is the overall Debian translation statistics[6], where you can find information about the translation status of files found in an actual package. This is similar to Fedora's Translation Status page[7], but unfortunately different, at this stage, in that we don't yet have a common (unlike it) there is no common infrastructure to facilitate the translation process.

An effort to translate the package descriptions started quite some time ago, even though very few tools currently support these translated descriptions (i.e. only APT can use them, when configured correctly). Maintainers do not need to do anything for this to work, and the translators should use the DDTP. For more information see Section 15.

[2] http://www.debian.org/doc/manuals/intro-i18n
[3] http://translation.sourceforge.net/
[4] http://developer.gnome.org/projects/gtp/
[5] http://i18n.kde.org/
[6] http://www.debian.org/intl/l10n/
[7] http://i18n.redhat.com/cgi-bin/i18n-status

For debconf templates, maintainers should use the po-debconf package to help translators work more effectively. Some statistics of the integration of debconf translations in packages can be found on the overall Debian translation statistics[8] pages. For more information, see Section 15

For web pages, each l10n team has access to the appropiate CVS, and the statistics are available at the overall Debian translation statistics site. For more information see Section 15.

For general documentation about Debian, the process is more or less the same as for the web pages (the translators typically need access to the CVS or SVN repository holding the documentation files), but there is no statistics pages yet, except for a few specific projects. Central information is definitely needed. For more information see Section 15.

For package-specific documentation (man pages, info documents, and information in other formats, like XML/Docbook, HTML and PDF), almost everything has yet to be done. The KDE and GNOME projects handles translation of their documentation in the same way as their program messages, but there is no specific way to handle translation of documentation in Debian project-wide[9]. The translation of Debian-specific man pages was initially handled within the manpages module[10] of the DDP CVS repository, but in some packages, manpages are handled just like generic documentation. For more information on how manpages translations are handled, see Section 15.

I18N & L10N FAQ for maintainers This is a list of issues that Debian package maintainers may face concerning I18n and L10n. While reading this, keep in mind that there is not yet any real consensus on those points within Debian, and that they are simply helpful advice. If you have a better solution to a given issue, or if you disagree on some points, feel free to provide your feedback, so that this document can be enhanced.

How do I get a given text translated? For translate package description or debconf templates, you have do not need to do anything at all. The DDTP infrastructure will send the translatable descriptions volunteers, and process the resulting translations with no need for you to interact. Addition of debconf templates is followed by translation teams that will send you new translations for them through the Bug Tracking System (you can contact debian-i18n, however, if you would like translations before releasing a new package version).

For all other material (PO files for applications, man pages or other documentation), the best solution is to make your text available somewhere on the Internet, and ask the translators on the debian-i18n mailing list to translate your file(s) into their different languages. The translation team leaders, at least, are subscribed to this list, and they will take care of the translation and of the quality-assurance processes. Once this is complete, your translated material will be emailed to you. You may receive queries about context for specific points (inserting context in your document will always ensure a better quality of translation), and get typo reports as a bonus feature.

How do I get a given translation reviewed? From time to time, somebody might translate some texts included in your package and will ask you to include it when you release. Understandably, you want to know if this translation is up to standard. You don't want to be releasing a campaign speech or a dirty limerick instead. Therefore, you can send the document to the Debian l10n mailing list for that language, asking for a review. Once that is complete, you can feel more confident in the quality of that translation, and can include it in your package with pride. You may also have introduced a new translator to that language team: a valuable resource for Debian.

How do I get a given translation updated? If you already have some translations of a given text, each time you update the original, you should also politely ask the previous translator to update his/her work, to bring that translation up to date with regard to the current original text. Keep in mind that this task takes time, just as it took you time to update the original: allow for at least one week to get your translation updated, reviewed and returned.

If the translator doesn't respond for some reason, please contact the Debian l10n mailling list for that language. The team-leader should respond promptly. If you don't hear back from the team, please email the Debian i18n mailing list, so it can follow this up or provide alternate contacts for the team. If, somehow, the translation doesn't get done in time (and this is rare, when you follow these procedures), don't forget to put a warning [11] in the translated document, stating that the translation is outdated, so the reader should also refer to the original document if possible.

You should avoid removing a translation completely, simply because it is outdated. An old document is usually better than no document at all for many users. Old translations might also include glossary information that might be useful for future reviews and you never know when a new translator will use the old translation to provide an updated translation (which is typically faster than writting a new translation from scratch).

[8] `http://www.debian.org/intl/l10n/`

[9] The Translation Project and GNU are curently running a pilot project on the manpage translation (and distribution) process which we may be able to implement.

[10] `http://cvs.debian.org/manpages/?cvsroot=debian-doc`

[11] If the translation is managed through gettext you do not need to do this as the gettext tools will prevent the user from seeing translations that have not been updated.

How do I handle a bug report concerning a translation? These reports should really go to the translation team, whose contact details are shown in all PO file headers, and should be shown on all documentation. It's a good idea to advise users to send any i18n bug reports straight to language teams[12]. When you do get them, please mark the bug as "forwarded to upstream", and forward it both to the previous translator and to his/her language team (using the corresponding debian-l10n-XXX mailing list).

How can I (as a maintainer) help the translation effort? You can eliminate a lot of wasted effort, and confusion, by organizing your end of the l10n process. Choose carefully what you want to translate (Debconf messages are always translated, but you might want to have additional items translated), and integrate translation with your release schedule (of either packages or new upstream versions). Allow for a string freeze before a release or package upload: this gives translators time to complete and review translations. Plan to request translations on the debian-i18n mailing list, at a realistic point before release, then again when you announce the string freeze. Communicate readily with debian-i18n and your translation teams: keep the information flowing.

If you are the maintainer of a popular package which might get translations after a release, you might get updated translations for upstream content (program messages and documentation) as bug reports or see changes in upstream's revision control system after you have uploaded a new version. Instead of sitting on the translations waiting for upstream to add them and making a release, you can make this translations available in the Debian package inmediately. This ensures that Debian users will see the translations as soon as they are available and it also boosts the morale of translators which see that their work is made available to users. Translators might get frustated if they send you updated translations for a package (even after sending them to upstream or updated upstream's repository) and do not see them being included in the Debian package until upstream decides to make a new upstream release. Unlike patches for source code, which might introduce regressions and instability, changes to PO files or new translated manpages do not affect the program itself and can be safely added.

I18N & L10N FAQ for translators While reading this, please keep in mind that there is no general procedure within Debian concerning those points, yet, and that in any case, the most effective way to resolve any issue is to work with your language teams. Language teams are set up to ensure an integrated approach to translation for that language, and that integration includes interfacing with developers.

How can I help the translation effort? If you are able to join the translation effort, welcome! We appreciate your effort. As a quick guide: decide what you want to translate, make sure that nobody is already working on it (using your debian-l10n-XXX mailing list), translate it, get it reviewed by other native speaker on your l10n mailing list, and provide it to the maintainer of the package (see next point). The Debian i18n web pages, especially the Translation HowTo, can give you more information.

How can I provide a translation for inclusion in a package? Most importantly, make sure that your translation is correct (by asking for review on your l10n mailing list) before providing it for inclusion. It will save time for everyone, and avoid the chaos resulting in having several versions of the same document in different bug reports. Translation teams are capable of quality work, and that is the reputation we want for Debian i18n.

The recommended way to submit translations is to file a regular bug against the package, with the translation file attached. Make sure to use the 'PATCH' tag, and to not use a gravity higher than 'wishlist', since lack of translations doesn't actually prevent a program from running[13]. For more information please read Section 15.

Best current practice concerning l10n

- As a maintainer, never edit the translations in any way (even to reformat the layout) without asking on the corresponding l10n mailing list. You risk, for example, breaking the encoding of the file by doing so. Moreover, what you consider as an error can be right (or even needed) in the given language. Trust the expertise of translators these areas.
- As a translator, if you find an error in the original text, please report it (by sending in another bug report, also of "wishlist" priority). Often, translators are the most attentive readers of a given text, and if they don't report the errors they find, nobody will. Also, it makes your job easier, if the original text is correct.
- In any case, remember that the major issue with l10n is that it requires several people to cooperate effectively, and that it is all too easy to start a flamewar about small problems, because of a misunderstanding. Avoid misunderstandings by being as clear as you can, using straight-forward language (avoid idiom, for example) and not

[12] Upstream users can add a link on their site, to the Debian language teams listing to prevent getting these mails
[13] It might, however, prevent people from using it

making assumptions. If you do run into difficulties, please ask for help on the corresponding l10n mailing list, on debian-i18n, or even on debian-devel.

- Essentially, cooperation can only be achieved through *mutual respect*. We are all working hard in the same project: we achieve so much more when we work together.

I18n/L10n projects in Debian

Debian has different ongoing processes, to facilitate the internationalisation of the software it develops, as well as to assist translators in working on the localisation of content produced for Debian. This chapter explains the different i18n and l10n ongoing projects in Debian, and describes how they work.

Translation of the Debian website

A project website is so often used as the primary information source for both the users of a program, and for people who want to learn about it but don't use it yet. Information on a project's website is usually complementary to the documentation available through other means. This website information is also typically more up-to-date than the documentation provided when the software was installed, or the distribution media in which it provided was purchased.

This gives the website some priority in the internationalisation and localisation process of a universal project. It's out there, it's current, and it's being viewed: so it is translate, to make it accessible to many more people. A website is a moving target for translation, requiring frequent updates, but it's a key access point for users (both existing and future).

In order to present translated information to the user, the Debian project website uses *Content Negotiation* technology (more specifically, Apache's content negotiation[14]) that is based on the user's web browser providing information on chosen languages (*Preferred-Language* HTTP header). The user may set this language preference individually in the browser, or simply ask the browser to follow the system preferences. Content negotiation, in this case, means that the website software can present a copy of the content translated to the user's most effective language (if a translated copy is available). More information on content negotiation is available on the Debian website's description of content negotiation[15]. Translated websites will typically include a "language bar" which shows in which languages that site is available.

In this way, server administrators will manage both the original document (typically in English) and multiple translations of that document. Content negotiation takes care of the task of looking for the document for a given URL in the nominated language, either showing it if it is available, or defaulting back to the original page.

Translations for the Debian website are being created for 33 world languages, although only 10 language temas have more than 500 pages translated out of over 3000 available pages, and only 5 teams have translated more than 50% of the site. It is a very large and demanding task, but one we assign priority, so there is a great deal of translation activity around the Debian website. New languages will continue to be added, and further pages translated.

The web server is based on content managed by the **wml** program ,which allows separation of templates and content, and also provides a mechanism to generate content within the pages on *compile*. Wml is to HTML what a C source is to its object code.

Most of the information on the web server is available in a directory tree, handled through CVS (at cvs.debian.org). In that tree, there is a directory for every language in which the website is translated. In principle, all the languages follow the directory hierarchy defined in the original (English) pages although, depending on the actual content translated, the contents of each subdirectory within a language might vary. There is also a specific location for translation teams' content which is not translations, but rather specific content written in that language, usually specific to that language/culture. The mechanism for generating the pages is based on **Makefile** in such a way that all the wml files in a given directory are compiled and published together. The use of included files means translators do not need to concern themselves with the content of the Makefile files.

Thanks to the independence of real content and aesthetic information, translators can simply take the original (English) wml files, move them to their own directories and translate them. The templates that generate the website itself need not be translated, unless specifically required. The templates use wml's internationalisation tools and **gettext** to extract and present the translatable information (including headers, footers, menus, buttons, both interactive and passive text) in all the pages for translation.

One of the main issues in ongoing translation is the need to be able to monitor changes in the original content. In particular, translators need to know which changes require updating translations. The web server has the same problem. Using content negotiation to detect a user's language settings, is only looking to see if a translation exists and can be displayed, and is not aware of the status of any translated content. So it's possible that users might be presented with

[14] `http://www.apache.org/docs/content-negotiation.html`
[15] `http://www.debian.org/intro/cn`

out-of-date information. This issue would not exist if the translation were removed when it was no longer current, since the web server would then simply provide the default language (English) to the user. However, many changes to the website are not fundamental changes, or are typo fixes to the original document, so it would seem senseless to provide the original document in these situations, since the translation (even if out of date) would still be useful to website readers who can't understand the original content.

To deal with this currency issue proactively, the website development team implemented a mechanism to detect out-of-date translations, based on *translation headers*. These headers are included in translated documents, describing which original file they translated, and specifying the CVS revision used. The use of this header makes it possible to introduce several additional tools to facilitate effective translations:

- an automatic program that can be run in the directory tree to provide a translator with a list of the documents that have not been updated for a given language. This is done through comparison of the current CVS revision of the original file with the CVS revision that the translated document states it used as a base for the translation;
- a wml toolkit that, at the same time pages are compiled (once a day), checks if the wml file being handled is a translation or not, and if it is, if it is current. It uses this information to introduce a predefined text in the output document, that tells the reader if the translation is not up to date, and points to the original translation (the text varies based on how out-of-date the translation is). The same tools will also add a footer to all pages, to list the translations available not simply for the website in any degree, but for each specific page.

More information is available on the website development reference[16].

There are further tools, also based on this mechanism, that provide statistics related to the translation effort[17], including statistics on out-of-date translations[18][19].

Since translations have to follow the creation of the original content, they will typically lag slightly behind the translated content in an active website. This mechanism provides a way for translators to work on translations at their own pace, while ensuring users reading the content know how current it is, and have the option to switch to the original content.

Translation of Debian tools

Internationalisation of the Debian distribution also involves the internationalisation and translation of the tools developed within the distribution itself. This includes the translation of the installation system (**d-i**), of the package management system tools (**dpkg**, **dselect**, **apt**, **aptitude**), of desktop menu management (**menu**) and of other Debian tools such as **debconf**.

Without proper internationalisation or translation of these tools, the majority of current and potential users will not be able to use or manage the Debian GNU/Linux operating system effectively.

The following sections describe the efforts in internationalisation and in the translation of different parts of the Debian GNU/Linux distribution within the Debian project itself

Debconf translation Debconf translation covers the translation of all interactions with the system administrator while installing packages (or installing the entire system).

Maintainers' tasks - internationalisation

The Developer's reference[20] recommends using the debconf protocol for user interaction. This presents information in a standardised way, easier for the user to follow and understand, and also allows more effective localisation of user interaction. Allowing debconf templates to be translatable is also a recommended practice[21].

This requirement could even become mandatory in the future. This was mentioned for Etch but unfortunately no-one followed through, by proposing the required changes to the policy for this to take place. Debconf, untranslated, is a bottleneck between the users and your software. Translate it, and it's an access conduit.

Debconf translations involve the **po-debconf** package which has now become a standard in Debian. Even though not mandated by the policy, all packages using **debconf** for user interaction should use **po-debconf**. It automates a number

[16] `http://www.debian.org/devel/website/translating`
[17] `http://www.debian.org/devel/website/stats/`
[18] `http://people.debian.org/~peterk/outdated/`
[19] Translations that are out-of-date for more than six months are automatically removed, regardless of the number of revisions they are behind. For more information read `http://lists.debian.org/debian-www/2004/01/msg00323.html`
[20] `http://www.debian.org/doc/manuals/developers-reference/`
[21] `http://www.debian.org/doc/manuals/developers-reference/ch-best-pkging-practices.en.html#s-bpp-i18n`

of the repetitive but essential tasks, and ensures that translators receive update notices for your templates. This is best practice in the field, ensuring a much higher translation rate.

The details of **po-debconf** are well covered in its man page (**po-debconf(7)**). In short, strings (Description, Choices and Default fields) should be prepended with the underscore character in the debconf templates file. If you do so, they will be included automatically in the list of translatable material.

The **debconf-updatepo** program gathers all the translatable material in a Portable Object Template (POT) file located under `debian/po` in the package build tree. This templates file is then used as a reference by translators to create a Portable Object file for their languages, using the language ISO 639[22] code.

Consistency of style Users will often see a series of several debconf screens in a row. A consistent style in both writing and presentation considerably improves the perception of professionalism in the overall Debian distribution.

Unfortunately, the original English strings very often lack this consistency, and despite some efforts in the Developer's Reference[23] to give maintainers advice about writing style for debconf templates, the manner of addressing users often varies wildly:

- use of the first person;
- use or not of interrogative form for input-style templates;
- repeated interrogative form in long and short description of templates;
- varying enumeration styles;
- use of specific references to some of the debconf interface-specific widgets or behaviour (talking about ”previous” or ”next” screens, assuming that boolean templates use Yes/No style questions, etc.).

In general, maintainers should follow translators' advice when they suggest improvements to original templates.

The French team, and more specifically Thomas Huriaux, has setup a dedicated page[24] which automatically tracks down the most common ”errors” with regards to this recommendation.

Repetitive strings A great deal of user interaction in debconf templates involves similar questions or input, such as web server configuration for packages that provide web services, database interactions for RDBMS-related packages or packages that use databases, basic identification, etc.

To avoid such repetition and the wasted time associated with it, packages aimed at providing common methods, as well as common debconf templates for such use have recently appeared in the Debian distribution. The **dbconfig-common** package is one of them.

Maintainers are encouraged to use these packages, and to help improve them for better quality and efficiency. The use of the debconf *register* commands is also encouraged in some cases, when a package needs to use strings or templates provided by another package.

Translation maintenance - localisation assistants The part of this paper which deals with i18n/l10n of Debian specific programs(see Section 15) introduces the concept of ”localisation assistants”.

As the maintenance of debconf translations is usually very simple, this paper does not enforce the use of localisation assistants, except in the case of packages with a high number of templates and/or packages listed a top priority packages for translators (packages which belong to the Debian Installer ”levels” packages, or very popular packages).

However, in any situation where the maintenance of debconf translations becomes a hassle for them, package maintainers are encouraged to cooperate with a localisation assistant. The debian-i18n mailing list[25] is the entry point that should be used to ”recruit” localisation assistants.

Translation work - localisation

Getting translation material Although everyone can access debconf translation material by downloading each package source and looking in `debian/po`, translators are encouraged to use links from the translation statistics pages[26].

[22] `http://www.loc.gov/standards/iso639-2/englangn.html`
[23] `http://www.debian.org/doc/developers-reference/ch-best-pkging-practices.fr.html#s-bpp-config-mgmt`
[24] `http://haydn.debian.org/~thuriaux-guest/templates/templates_by_packages.html`
[25] `http://lists.debian.org/debian-i18n`
[26] `http://www.debian.org/intl/l10n/po-debconf/`

Priorities Although debconf templates are fairly small, there are over 695 individual debconf templates, a fairly large and repetitive task for translators. The number of Debian packages with translatable material for debconf is pretty important. As of this writing these 695 packages with translatable debconf material add up to about 10000 strings to translate.

Recent changes to the translation statistics pages now allow sorting packages by their popularity, using data from the **popularity-contest** package. This was achieved to avoid translators spending their effective time on rarely-used packages, and to ensure commonly-used packages get translated first.

Statistics Statistics for debconf translations are gathered on the Debian l10n status pages[27] and are separated from the translation statistics for individual programs. These pages also give access to the translation material collected by the statistics robot operated by the Debian i18n team.

Translation Translation usually involves copying the `templates.pot` file to `<code>.po` (where *code* is the language code it is being translated to), filling in this file's header with appropriate information, then using a Portable object editing tool (or any text editor) to enter the translations for all the strings.

Checking translations Translators can test the debconf PO files by using the **podebconf-display-po** tool from the **po-debconf** package. They will need to install this package on their system.

A locale for the tested language must be built on the system (which can be achieved by "dpkg-reconfigure locales") and the LC_MESSAGES variable should be set to this locale.

It is recommended to test the debconf templates by using the debconf dialog interface in a 80x25 screen.

podebconf-display-po has a few quirks, especially when some strings are shared among several templates, so it should not be relied on blindly. Despite this, it has already proven extremely helpful in detecting formatting issues.

Style consistency Translations need a consistent writing style, just as the original strings do, to communicate effectively. This information, and the way it is presented, also functions as the public face of that application or document, and of the overall project.

While it is true that, as yet, the original English strings in many packages may lack this professional consistency, translators can greatly help package maintainers a great deal in improving accuracy and style. They can report syntax and style errors as bug reports against the relevant packages. Ultimately, this partnership between package maintainers and translators not only produces an application accessible by many more people, but also a higher overall quality of presentation.

Translators should aim for consistency in their own translation work. Even if the original strings do not follow writing-style recommendations, translators should adapt their translation to be consistent in their language, to create a high-quality translation as an achievement in its own right.

Translation teams, and the QA work they do, have indeed a great role to play in this overall improvement of presentation consistency. They are capable not only of processing information for translation, but also of reviewing it as information. Through this additional review process, the quality of information presentation in debconf templates should continue to improve.

Sending new translations and translation updates to maintainers Translators should send debconf translation updates to maintainers via the Bug Tracking System[28], even when the update is requested by a maintainer using an automated tool to call for updates. Such bug reports should use the "l10n" and "patch" tags, and severity "wishlist"

They should be filed against the *source* package rather than against a binary package.

The use of the reportbug utility is recommended.

A standardised bug title is recommended: "¡package¿: [intl:¡code¿] ¡Language¿ debconf templates translation", where ¡package¿ is the source package name, ¡code¿ the language ISO code and ¡Language¿ is the language name in English.

Here's an example of reporting the French translation of the **geneweb** package. Of course, in the example below, **geneweb** should be replaced by the appropriate package name:

```
bubulle@mykerinos:~/tmp> reportbug geneweb
*** Welcome to reportbug.  Use ? for help at prompts. ***
Using 'Christian Perrier <bubulle@debian.org>' as your from address.
Detected character set: UTF-8
Please change your locale if this is incorrect.
```

[27] http://www.debian.org/intl/l10n/po-debconf/rank
[28] http://bugs.debian.org/

```
Getting status for geneweb...
Checking for newer versions at packages.debian.org...
Will send report to Debian (per lsb_release).
Querying Debian BTS for reports on geneweb (source)...
14 bug reports found:

(snip all bug reports for geneweb)

(1-14/14) Is the bug you found listed above [y|N|m|r|q|s|f|?]? n
Maintainer for geneweb is 'Christian Perrier <bubulle@debian.org>'.
Looking up dependencies of geneweb...
*** The following debconf settings were detected:
  geneweb/run_mode: Always on
  geneweb/remainingdir:
* geneweb/remove_databases: false
  geneweb/port: 2317
* geneweb/lang: French
Include these settings in your report [Y|n|q|?]? n

Please briefly describe your problem (you can elaborate in a moment; an
empty response will stop reportbug). This should be a concise summary of
what is wrong with the package, for example, "fails to send email" or
"does not start with -q option specified."
> [INTL:fr] French debconf templates translation
Rewriting subject to 'geneweb: [INTL:fr] French debconf templates
translation'

Enter any additional addresses this report should be sent to; press
ENTER after each address. Press ENTER on a blank line to continue.
>

How would you rate the severity of this problem or report?

1 critical        makes unrelated software on the system (or the whole system)
                  break, or causes serious data loss, or introduces a security
                  hole on systems where you install the package.
2 grave           makes the package in question unusable by most or all users,
                  or causes data loss, or introduces a security hole allowing
                  access to the accounts of users who use the package.
3 serious         is a severe violation of Debian policy (that is, the problem
                  is a violation of a 'must' or 'required' directive); may or
                  may not affect the usability of the package. Note that non-
                  severe policy violations may be 'normal,' 'minor,' or
                  'wishlist' bugs. (Package maintainers may also designate
                  other bugs as 'serious' and thus release-critical; however,
                  end users should not do so.)
4 important       a bug which has a major effect on the usability of a package,
                  without rendering it completely unusable to everyone.
5 does-not-build  a bug that stops the package from being built from source.
                  (This is a 'virtual severity'.)
6 normal          a bug that does not undermine the usability of the whole
                  package; for example, a problem with a particular option
                  or menu item.
7 minor           things like spelling mistakes and other minor cosmetic errors
                  that do not affect the core functionality of the package.
8 wishlist        suggestions and requests for new features.

Please select a severity level: [normal] 8
```

```
Do any of the following apply to this report?

1 l10n          This bug reports a localization/internationalization issue.

2 patch         You are including a patch to fix this problem.

3 experimental  This bug only applies to a package in the experimental branch
                 of Debian.

4 none

Please select tags: (one at a time) [none] 1
- selected: l10n
Please select tags: (one at a time) [done] 2
- selected: l10n patch
Please select tags: (one at a time) [done]

(snip editor being spawned)


Report will be sent to "Debian Bug Tracking System" <submit@bugs.debian.org>
Submit this report on geneweb (e to edit) [Y|n|a|c|e|i|l|m|p|q|?]? a
Choose a file to attach: /home/bubulle/tmp/fr.po
Report will be sent to "Debian Bug Tracking System" <submit@bugs.debian.org>
Attachments:
 /home/bubulle/tmp/fr.po
Submit this report on geneweb (e to edit) [Y|n|a|c|e|i|l|m|p|q|?]? y
```

DDTP Another translation project is the translation of the descriptions of the software packages offerred for the Debian GNU/Linux OS. This project is named the *Debian Description Translation Project* (DDTP).

Each package shipped in Debian GNU/Linux is provided with two descriptions: a short description (less than 80 characters) and a long one (of variable length). The first one describes briefly what the package does and the second one describes its main functionalities, characteristics, differences with other software, etc. This information is vital for users who are looking for a given functionality within the (huge) quantity of software provided in the Debian distribution. The package management tools include interfaces to do word (or regular expression) searches through these descriptions.

However, the fact that all these descriptions are only available in the English language makes it difficult for users that have already installed the system to look for new software they might want. Users not fluent in English might not be capable to define their needs in a foreign language to look for the specific software they are looking for.

The description translation project started in order to solve this problem for end users. This project was started initially by two different work groups in the year 2002, resulting in two different management systems (one based on e-mail and another based on a web application[29]. The framework was based on Perl scripts that collected descriptions and provided an e-mail interface for translators (translators could request new translation through email and submit them using the same interface). All translations are stored in a simple file with a database holding the status of the translation (new, needs to be reviewed, out of date, etc.). Most notably, the framework did not use PO files for translations which got it many objections from some translation teams.

The translations from the web based interface one of these systems have later been merged on with the other one. The project was used by different translation teams. When Debian systems were compromised[30] in November 2003 the main framework was disconnected pending a review of the source code.

The Brazilian team (Otavio Salvador) later developed patches for **apt** to support the use of the translated descriptions in it. This patch (apt-ddtp[31]) was ported to the 0.6 code base, with the help of some other developers, including Michael Vogt. Later, in April 2005, an Alioth project[32] was started and the original source code of the translation framework was moved to a SVN repository[33].

As of this writing the DDTP project does not have an interface to submit translations. The translations already done for descriptions can be downloaded from `http://ddtp.debian.org/`, but they are not being updated.

[29] This web application was developed by members of the Spanish translation team and available at `http://www.laespiral.org/`
[30] `http://www.debian.org/News/2003/20031121`
[31] `http://people.ubuntu.com/~mvo/arch/ubuntu/apt--ddtp--0/`
[32] `http://alioth.debian.org/projects/ddtp/`
[33] `http://svn.debian.org/wsvn/ddtp/`

This work, however, has been reused in Ubuntu. Currently, the Ubuntu package descriptions are imported into Ubuntu's translation framework: Rosetta[34] using the po/pot extraction/assemble tools developed by Michael Vogt, and available at michael.vogt@ubuntu.com--2005/apt-ddtp-tools--main--0[35]. The tools converted the Packages file to a pot file that is then imported into Rosetta. Rosetta then generates the PO files from the translations that can be downloaded and converted into a Translations-$lang file that apt-ddtp understands.

Translation of installed systems

A common need for users installing a Debian system is to end up with a fully localised system that does not need to be set up in order to be useful for users who require a given locale for being supported. Besides having a fully I18n Debian Installer there are several things that help set up an internationalised Debian system.

Localization-config **Localization-config** is a tool to setup the configuration of programs in order to use a default language for the system as defined by the system administrator. This tool was initially written by Konstantinos Margaritis, developed for Skolelinux and was later integrated with Debian and uploaded in August 2004. It was later integrated in the Debian installer for sarge (through **base-config**).

This tool tries to ease the way that administrators (or, even, the Debian Installer) set up a fully localised system. In a system, localisation begins by defining the preferred locale but there are many tools that need to be specifically configured to use a given locale. This tool can run after and before package installation to adjust its configuration to use a given language through the use of a collection of scripts. Each of the available scripts adjust the configuration of a single package for a number of languages. Please note that this tool is targeted for system configuration. Users wishing to change their own locale (but not the system's) should use **set-language-env** (in the **language-env** package).

In order to use this tool, the system administrator have to call one script, **update-locale-config**, providing the desired locale as a parameter. This script calls all other scripts in `/usr/lib/localization-config`. If the -p flag is used, the *.preinst scripts are called, else (default) the *.postinst scripts are called. The preinst scripts are supposed to be run when you need to do configuration of a package before it is installed (main use is by preseeding debconf values of the package). The postinst scripts are there when a package does not support this kind of preconfiguration (such KDE, gdm, etc.) and has to be configured by modifying its configuration file.

In order to prevent modifying new configuration file formats the scripts first checks the version of the package it should configure, in three different ways:

- check the version of the installed package.
- check the version of the package available available in the configured repository (that is, what is the version of the package you would install if you use **apt**).
- if both of the checks above fail, try getting the version from `/etc/debian_version`. This method should never be called if everything works as expected.

The version of the package is used to define version maps, since some versions might need to be configured in slightly different ways. The wrapper scripts will call the appropriate script under the proper folder (woody, sarge, etc) and do the actual configuration.

Localization-config currently configures:

- Dictionaries-common (**ispell**): preseeds the preferred dictionary;
- fontconfig: implement fonts substitution (currently only for Greek);
- Gdm: sets the default language for sessions;
- KDE: modifies the language settings of `/etc/kde3/system.kdeglobals` and `/etc/kde3/kdm/kdmrc`. It can also modify the settings of the **ktouch** program by adjusting `/etc/kde3/ktouchrc`;
- Links: Sets the preferred language for browsing by modifying `/etc/links.cfg`;
- Ltsp: Sets the X keyboard map by modifying `/opt/ltsp/i386/etc/lts.conf`;
- Lynx: Sets the preferred language for browsing by modifying `/etc/lynx.cfg`;
- Mozilla: Sets the preferred language for browsing by setting the UserAgent locale and accepted languages at `/etc/mozilla/prefs.js`;

[34] `http://launchpad.ubuntu.com/rosetta`
[35] `http://people.ubuntu.com/~mvo/arch/ubuntu/apt-ddtp-tools--main--0/`

- Xfree86 and Xorg: preseeds the character set for the language.

Localization-config is still work in progress and members of the different i18n teams still need to review the available scripts in order to integrate their own languages in the installation process. Moreover, due to the changes introduced in the Debian Installer early in 2006 (in which base-config was removed and introduced into the first stage of the installed), this package needs to be re-integrated into the *etch* installer.

New i18n scripts need to be written also for programs such as: **locale-purge** (`/etc/locale.nopurge`, TeX/LaTeX (including hyphenation rules defined at `/etc/texmf/language.dat`), console settings (including `/etc/inputrc` and the console keymap by preseeding **console-common**), Mutt (locale and charset at `/etc/Muttrc`), Mozilla's Firebird (same values as those configured already for Mozilla), etc.

Language tasks The Debian task selection tool, called **tasksel** is run in every new system installation and allows to define a set of packages aimed for specific purposes.

One widely developed used of tasksel are "language tasks". These are tasks that are installed depending on the installation language. They should feature sets of packages that are specific to the related language.

Sets of language tasks

There are currently two language tasks per language: one ¡language¿ task which depends on nothing but standard packages, and one ¡language¿-desktop task which depends on the standard desktop task.

Packages in language tasks can include localisation packages for software that are installed by other tasks or by the standard system. For instance, the ¡language¿-desktop tasks install localisation packages for the packages that are part of the standard desktop task (kde-i18n-*, openoffice.org-l10n-*, etc.).

These tasks can also include packages that are needed for the rendering of the given language: TTF or Postscript fonts for ¡language¿-desktop tasks, console fonts for ¡language¿ tasks, etc.

Maintenance of language tasks

Adding new language tasks should be partly automated by the tasksel maintenance team, i.e. the Debian Installer team. As soon as a new languages appears in the D-I supported languages, an equivalent language task should be added, if at least one specific package motivates it (for instance a localisation package for on of the packages that are part of the desktop tasks.

Translation teams and translators should also be active by proposing enhancement to tasks related to their language.

New tasks should be reported as wishlist bug reports again the tasksel package.

Translation packages This section will be completed in further releases of this paper

Translation of documentation

Users of an operating system require, in order to be able to use it, both online and offline documentation.

In UNIX systems, online documentation is typically handled through manual pages (retrieved through the use of **man**)[36]. In Desktop environments both KDE and GNOME provide online help systems.

Offline documentation includes documents such as the Debian Installation Manual, the Debian Reference Guide, or the Security Debian Manual. These manuals, developed by the Debian Documentation Project[37] (DDP), are typically printed by users and, sometimes, read from other (non-Debian) systems at the project's website. Some of the manuals are included in the official CD set and most of them are also available through Debian packages.

Obviously, users not proficient with English will prefer documentation in a language they're familiar with. This is why translation of documentation is needed.

SGML/XML documentation The *DDP* project develops documentation for Debian written, primarily, in SGML format. The first documentation produced by the project was written in a variant of SGML called *debiandoc-sgml* for which tools were developed to convert the SGML documents into different formats (HTML for online viewing, PDF and PostSCript for printing, and simple text). The latest documentation written by members of the project or revisions of available documentation (such as the Debian Installation Manual) has been written using XML and more specifically Docbook-XML which is widely used by many free software projects[38].

The translation of documentation written in SGML or XML format, however, faces some initial issues:

[36] The GNU project prefers the use of **info** documentation but most upstream developers just provide manpages.

[37] `http://www.debian.org/doc/ddp`

[38] Including distributions such as Red Hat GNU/Linux, or the Linux Documentation Project[39]

- SGML documents are usually written in a single (big) file. Since translation teams are typically created in order to handle (small) documents this means that the translation coordinator has to break up the original file in chunks;
- Translators cannot directly use the tools they are used to such as tools intended to work on PO files;
- Automatic publication of documentation in HTML format of documentation needs to be adapted so that they are capable of publishing both the original document and the translations without one conflicting with each other;
- Automatic publication of documentation in PS/PDF needs to be able to handle non-European character sets (i.e. specific fonts) properly;
- Tracking of changes and updates in SGML/XML documents needs to be done through source diffs (which requires access to a revision control system). If a single file is being used, tracking differences will be more difficult;
- Diffing tools handle poorly the fact that text can be wrapped to different column lengths and paragraphs might be ”moved” around without actually changing context.

In order to overcome these issues, the document writers, in cooperation with translators, have introduced significant changes to how documentation is written:

- SGML documents are broken up into smaller files (typically one file per chapter);
- New tools have been introduced to be able to convert SGML/XML documents into PO files: **po-debiandoc** (now deprecated), **po4a** (see Section 15) and **poxml**;
- New tools have been introduced to be able to detect when files within a document have changed: **doc-check** (see Section 15) and track the specific changes through the revision control system;
- The publishing toolset (based around `Makefiles`) has been adapted in order to build and publish both the original file and the available translations.

Currently, the Debian Documentation Project uses the CVS server at cvs.debian.org[40]. Documentation is compiled through a set of Makefiles and published on the official project web site[41], which updates its copy of the CVS repository and compiles all the documentation daily. The CVS server at cvs.debian.org holds most (but not all) of the documentation available. The most notable exception is the *Debian Installer Manual* which is available in the SVN repository of the Debian Installer project on Alioth[42] (web access to the SVN repository[43]). The development version of the Debian Installation Guide is available at the Debian Installer project's web pages[44] also on Alioth.

Translators either get access to the CVS or use the original documentation maintainer as a proxy to publish the information in the CVS. When a translation is added to an available document the maintainer typically needs to update the *LANGS* (or *LANGUAGES*) variable in the document's `Makefile` in order to tell the publication system to also build copies for that language. If the added translation builds, it should be available in the Debian website after the next daily build.

In order to track the changes of documentation and when a translation needs to be updated, document maintainers and translators use the **doc-check** tool. See Section 15.

Debian manpages translation There are mainly two types of manpages provided within the Debian operating system: those packaged with upstream software (such as binutils or gcc) and those written for Debian tools provided in Debian-only packages. The translation of manpages of upstream software are typically provided either within the package itself or within `manpages-XX` package (where `XX` is a given language codename) the case of the translation of the Linux manpages[45].

Manpages packages

At the time of this writing the following translation manpages packages are available:

- manpages-de and manpages-de-dev - German manpages
- manpages-es and manpages-es-extra - Spanish man pages

[40] The web interface can be accessed at `http://cvs.debian.org/ddp/manuals.sgml/?root=debian-doc`.
[41] `http://www.debian.org`
[42] `svn://svn.d-i.alioth.debian.org/svn/d-i/trunk/manual`
[43] `http://svn.debian.org/wsvn/d-i/trunk/manual/?rev=0&sc=0`
[44] `http://d-i.alioth.debian.org/manual/`
[45] `http://www.kernel.org/pub/linux/docs/manpages/`

- manpages-fi - Finnish man pages
- manpages-fr - French version of the manual pages
- manpages-hu - Hungarian manpages
- manpages-it - Italian man pages
- manpages-ja and manpages-ja-dev - Japanese version of the manual pages
- manpages-ko - Korean version of the manual pages
- manpages-nl - Dutch manpages
- manpages-pl - Polish man pages
- manpages-pt and manpages-pt-dev - Portuguese Versions of the Manual Pages
- manpages-ru - Russian translations of Linux manpages
- manpages-tr - Turkish version of the manual pages
- manpages-zh - Chinese manual pages

Those manpage packages contain, for the most part, the same manpages available in the `manpages` or `manpages-dev` packages although some packages contain extra manpages.

Users will see the translated manpages through the internationalisation mechanisms of the **man** command which will review, when asked to present a manpage, if a translation is available under `/usr/share/man/XX` (with XX being the language code of the user's environment[46]). Consequently, the Debian installation system will install both `manpages` and `manpages-XX` through a default installation if the user selects an specific language task for which manpages are available.

Issues with manpage translations

One of the main issues with manpages, however, is that there is no provisions in the **man** to detect when a translation is out of date. As a consequence, users reading translated manual pages might be reading out of date content that does not really apply to the latest version of the program's man page.

The translation project is also lacking a central web page where teams can see (at a glance) which manpages are available for translation, which translations are out of date, translated or untranslated and who is the last translator of the manpage.

Translation of Debian manpages

The translation of manpages specific to programs developed within the Debian project (such as the **dpkg** or **apt** tools) is a work that falls within the scope of the Debian translation teams.

The translation of manpages for Debian programs are included within the Debian package itself, which means that translators have to request the Debian maintainer to include the translation in it, once finished. Since Debian programs are typically managed through common revision control repositories available to Debian developers or contributors (either at cvs.debian.org or at alioth.debian.org) active translators of a Debian program will typically have access to those resources and will be able to commit directly into the source control zone were manpages are included. It is worthwhile noting, that it is also common for people active in the program translation to work on the translation of the manpage so that the translation of the program messages and options is consistent with the manual page itself.

In order to coordinate the translation of manpages and make it possible to track when the translation changes, the translation teams introduced the `manpages` in the CVS DDP area. This module includes several scripts in order to track manpages translations: **check_trans.pl** and **compare_files.pl**[47].

This module was introduce since translators did not have access to the CVS repository of the programs for which the translations were going to be made available. Consequently, the original manpages themselves could not be modified to include a translation control header to keep track whenever one was modified. In order to keep track of translation status the CVS module holds `INFO` with meta-data of translated documents including:

Manpage document's name in the CVS repository, may be different from the one in the source package. This is used as the document ID.

[46] As defined through the LANG variable

[47] There is an additional script, **gen_db.pl**, used to generate the wml files used for the translation coordination database in the web page area

Encoding Document's encoding.

Location Location of this document in the source package (this value is only set when the source package does contain this document).

Original Original ID in the `english/` directory.

Original-CVS-Revision CVS revision number of the original document on which a translation is based.

Translator Translator's name, is be used to send automatic notifications when a translated man page is outdated.

Original manpages are included in the `english/` directory of that CVS module by the translation teams and need to be updated manually when the original file is updated. Based on the meta-data information and the CVS revisions available for manpages the scripts can track when a manpage is outdated and notify the translator in charge of it.

Unfortunately, this mechanism, initially developed by the French translation team and used by other teams, is not being maintained. There have been no updates in the English manpages for two years and translations have not been updated there either.

FIXME: Describe use of po4a. The CVS-DDP module is not that much used anymore since most translators are now in the packages themselves...

Coordination of documentation translation Initially, some of the translation projects (French and Spanish) introduced their own documentation translation management system[48] in order to coordinate the translation of the Debian Documentation Project published manuals. This management system was based on a flat database that included the available documents in the DDTP system and the status of translations. With the use of Perl scripts, this database was converted into HTML files that were published on the website so that the translation team could see which documents were being worked on and who was coordinating the translation.

This system was not integrated with the document database provided by the DDP itself and the translation teams have, for a few years, made use of the translation robots (see Section 15) in order too coordinate translation of documents themselves.

i18n/l10n infrastructure in Debian

Po Translation statistics

The Debian web site features statistics web pages under `http://www.debian.org/intl/l10n`. These pages collect statistical data about po-debconf and programs translations.

The statistics are collected daily by a script run under a developer's account on people.debian.org (Denis Barbier account at the time of this writing). These statistics use material gathered by another script run under a developer's account on people.debian.org (Pierre Machard at the time of this writing).

The statistics pages help translators learn about areas that need work such as updates for existing translations or new packages/material needing translation work.

The pages also help translators to grab POT files and start working on new translations as well as PO files and work to complete them.

Even though this system has proven to be highly useful during last years, it still carries a few weaknesses that prevent calling it an overall i18n/l10n infrastructure:

- it depends on scripts running under individual developers accounts;
- it only gives statistics for the *unstable* branch of the distribution and thus does not allow to get statistics about l10n in testing during release preparation;
- it does not track down the status of the work by translation teams and does no point the reader to the existing translation teams. For that reason, teams have developed their own tracking work method (some parts will be detailed in Section 15)
- it is not linked to the bug tracking system and does not allow checking whether an incomplete or missing translation has a pending fix in the BTS (and for how long the fix is pending) or in the package development revision control system.

A way to go would be integrating all these needs in a more general i18n/l10n infrastructure.

[48] The Spanish management system (the status database has not been updated since January 2003) is available at `http://www.debian.org/international/spanish/ltcp/`.

Website translation statistics

Statistics for the translation of the Debian web site are collected in a different set of pages, run by different scripts.

This section will be completed in further releases of this paper

Debian installer translation statistics

In order to track down issues more closely, the Debian Installer (D-I) i18n coordinators have setup translation statistics pages which gather the status of l10n for all core D-I packages as well as packages defined as part of the D-I ”levels” of translation (FIXME: reference to DC5 talk).

These pages point to the individual packages RCS directories and archives. They give translators a more accurate view of work to do and immediately reflect applied changes.

They could be enhanced in a few ways:

- integrate them in the main web site. The status pages are generated by scripts running under Dennis Stampfer account. These scripts are not publicly visible and setting a new location for the pages would require Dennis expertise;
- revamp the pages to make the main page less verbose. There is currently a big amount of information on this page and it should be redesigned as a whole web site...or integrated into a more general i18n infrastructure;
- allow getting statistics about all components in their own RCS trees, in unstable and in testing. This would allow getting a better picture of the real l10n status for releases of D-I.

A way to go could actually be integrating these requirements in a more general i18n/l10n infrastructure.

Translation robots

One of the things that typically takes more time when coordinating translation projects is keeping track of what is each member of the team working on, and what is the precise status of each of the translations.

The most well known translation robot is the one used by the Translation Project[49]. This is an e-mail service that takes care of PO file submissions for translations registered within the project. It checks if files sent to it are receivable, that is, if a translator has filled the translation disclaimer [50]. This robot also calls **msgfmt** to see if the PO file is healthy. The robot also sends notices of updated PO files to the translation teams whenever translations need to be updated.

This translation robot however, only tracks PO files, and only PO files of those GNU projects registered with it. Bearing in mind that the requirements of the translations teams in Debian were different, the translation teams started writing translation robots to handle their own translation process. This work was started by members of the French team and then reused by other translation teams including Spanish, Dutch, Brazilian Portuguese, and German.

The translation coordination robot is an e-mail robot that ”listens” to the mails sent to the translation teams mailing lists (debian-l10n-XXXX) and looks for a set of pseudo-urls in the messages’ subjects. These pseudo-urls are composed of a translation status and a translation item (see Section 15). The robot takes this information to compose a list of items being translated, the status of the translation and the translator in charge of it.

This is a useful tool for both translation coordinators and members of the translation teams. Any member can, at a glance, see the status of a translation and help if help is needed (for translations or reviews). It also helps people detect translations that are stalled (a translator stated that they were going to work on it, but didn’t finish actually finish it).

Currently, there are several translation robots in place. First, there is a generic robot[51] that handles different mailing lists by crawling the web site information with a set of scripts[52]. This robot is used by most translation teams, although some teams use their own robot, currently active are the: Spanish team which uses its own robot, available at Spanish translation coordination robot[53], Dutch translation coordination robot[54], Catalonian translation coordination robot[55]. These last robots, instead of crawling the web site, use real e-mail addresses which are subscribed to the team’s mailing lists and handle messages in real time through procmail filters that filter this information to the translation robot’s status database.

[49] http://translation.sourceforge.net/

[50] Translators of the Translation Project have to send, through postal e-mail, a form that disclaims in writing by the translators, before being accepted for inclusion in the distribution. For more information, read http://translation.sourceforge.net/HTML/disclaim.html

[51] http://people.debian.org/~bertol/

[52] http://people.debian.org/~bertol/scripts/dl10n/

[53] http://www.debian.org.es/cgi-bin/l10n.cgi?team=es

[54] http://dutch.debian.net/

[55] http://ca.debian.net/

Translation stages Translations evolve through the following stages:

- First, a translation needs to be done when there is a new (untranslated) item or document. Typically, the translation team coordinator asks somebody in the team to work on it;
- Then, a member of the translation team answers by saying that (s)he will work on that item;
- Once the translator finishes the translation (s)he sends the item for review to the translation mailing list;
- After several reviews by peers, the translation is finally changed and a new version is sent for a final review;
- The translator then picks the final version and submits it to the upstream maintainer. This is sometimes a package maintainer, but it can also be somebody with access to the CVS where the translations are maintained. Typically, for some translations, this step involves translators using the Debian Bug Tracking System to contact the maintainers (see Section 15);
- After some time, upstream maintainers incorporate the provided translation in the package (or CVS) and the translation is considered ”published”. This is the final step for a translation, until the original translated item changes.

When the original item (document, wml or PO file) is modified, the cycle starts again. However, in that case, the last translator who worked on it is considered upstream's point of contact ((s)he should be contacted whenever there is a need to update the translation). The last translator is also typically considered as the maintainer for upstream's translations so there is no need to tell the translation team that (s)he will be updating the translation ((s)he is supposed to do it, as (s)he is in charge). Moreover, on many occasions, when the changes to the original item are few, there is not really a need to do a full review of the new translation by the translation team.

Pseudo-urls used by the robot A pseudo-url is made of the following:

```
[<state>] <type>://<package>/<file>
```

These pseudo-urls are used in the mail Subject: to help the robot distinguish which mails need to be handled by it and which mails are part of the mailing list discussion.

The contents of the pseudo-url are:

state The state the translation is in, for more information see Section 15.

type The type of item being translated. The translation coordination robot accepts the following item as valid types in the pseudo-url to indicate a translatable item: po-debconf, debian-installer, po, man or wml (webwml is deprecated, wml should by used instead).

package the name of the package where the document came from. www.debian.org is used for the wml files of the Debian web site cvs.

file the filename of the document, it can contain other information such as the path to the file or the section for a manpage, so no other document in the same package should be referred the same.

The structure of name depends on the chosen type. In principle it's just an identifier, but it's strongly recommended to follow the following rules:

- po-debconf://package-name/language.po
- po://package-name/path-in-sourcepackage/filename.po
- debian-installer://package-name/path-in-sourcepackage
- wml://www.debian.org/address_of_page
- man://package-name/section/subject

Translation states handled by the robot

The translation coordination robot can track what stage is a translation for any item through the use of the following keys in the 'state' part of the pseudo-url:

TAF ('Travail Faire', French for "translation to do", "taf" also means "work" in French slang) is sent to indicate that there is a document that needs to be worked on;

ITT (Intent To Translate) indicates that there is a translator that is planning on working on a given translation. This helps preventing translators to duplicate their work;

RFR (Request For Review) states than an initial translation is finished. The translator will attach the translation itself to the sent e-mail, for peer review. This key might be used more than once for the same item[56] if substantial changes have been made to the initial translation based on other's comments. Just like with CVS commit e-mails, translators expect a reply to these requests even if it's just to say 'The translation is OK.';

ITR (Intent To Review) a peer of the translation team notes that (s)he is working on a review of the translation and might take some (typically because the translation is large, or because the reviewer will not have time available until a given point in time) This is used to prevent the original translator to consider the translation as finished (send an LCFC, see below);

LCFC (Last Chance For Comments) tells the team that the translator considers the translation to be finished and has included the comments from the review process. (S)he is giving a last chance for peers to review before the translation is submitted upstream. Typically, it is sent when there are no no ITR's, discussion following the RFR has ended and it has been three days since the RFR was sent. Most translation teams don't allow translators to do this unless at least one member of the team has reviewed the translator's work;

BTS#¡bug number¿ (Bug Tracking System) tells the team that a bug has been open to submit the translation to the maintainer. This is useful, since the translation robot can then automatically start checking if the bug report is still open and updates the translation status accordingly;

FIX#¡bug number¿ (bug FIXed) notes that an open bug has been fixed already (useful if the translation robot missed it being closed);

DONE states that the translation has been finished and is now included upstream. This should be used in cases where no bug report is involved such as web site translations. Otherwise, the robot will handle DONE automatically by crawling the Bug Tracking System;

HOLD put a translation on hold, when the original version has changed but there is no need to update the translation, e.g. the translator knows other modifications will be done soon on the translation and they don't want someone else to update it too quickly.

Example of translation robot usage This is a typical example of the way the translation robots are currently used.

- A translator (T) wants to work on the PO-debconf translation of the exim4 package, instead of just saying so in natural language in the mailing list. (S)he sends this:

  ```
  Subject: [ITT] po-debconf://exim4
  ```

  The translation robot, when seeing that format in the mail's subject, processes the mail, retrieves the data (it is an ITT, of the po-debconf type, for the exim4 package, send by translator T on this date and time), stores it in a database. This information is presented whenever the translation status page is viewed (since it's driven by the database). The body of the mail is not processed, only the subject.

- Translator T completes the translation a few days later and wants people to review his/her work, so (s)he sends the following mail:

  ```
  Subject: [RFR] po-debconf://exim4
  ```

  The file to review should be attached to the mail[57]. The robot processes this mail, notes the status change in the database and extracts the file from the mail. The web pages can be used to retrieve the file itself.

[56] Some translation robots use RFR2 for subsequent reviews

[57] Some translation robots don't handle compressed files but most will handle MIME attachments

- After some reviews, translator T sends a mail with an LCFC, attaching the file (which is, again, parsed by the robot) and after a few days sends the document to the BTS and sends this mail to the list:

  ```
  Subject: [BTS#123456] po-debconf://exim4
  ```

  Once this is done, the translator's work is considered finished and the translation robot will, through a periodic job, review if the bug is closed in the bug tracking system. Whenever it is closed, it will be marked as DONE (and will be hidden from the view of the page after a month)

Throughout this process both members of the team subscribed to the list, new members which were not subscribed when the process started and the translation team coordinator have full access to the translation status (and its history) through the web application of the translation team robot.

Future changes for translation robots In the future, the different translation robots of the translation teams should be merged into one common database as part of Debian's infrastructure for translators. This would help prevent having robots coded in different ways and help that new features (such as handling compressed translations or testing PO files with **msgfmt**) need to be coded in each of the independent robots.

i18n/l10n tools in Debian

Generic tools: gettext

In order to be able to internationalise the user environment the GNU project developed a set of tools which are known as **gettext**. These tools make use of the **locale** definition that is a part of the POSIX.2 standard and is implemented in the GNU **libc**. These definitions include the basic tasks of representing currency formats, date or numbers. But they also include the definitions of sorting tasks and the classification of codes based on the user's culture (such as the order of the letters in the alphabet). This aspect of a user's environment only needs to be defined once since these definitions, once covered, do not typically vary throughout time (except when fixing bugs or introducing new languages).

The translation of messages according to the user's language in any given interface is, however, a more variable aspect of internationalisation of software. Not only messages that are shown to the user are translated usually, error messages, help of the options that the program use and any other message that a user might see at any point in time is suitable of translation.

The **Gettext** tool was developed within the GNU project between 1994 and 1995 by a diverse group of programmers. This tool helps the creation of programs that can be distributed with multiple message catalogs in different languages. In localised environments the programs will choose the message catalog most suited to the environment as defined by the user and present messages in a given language.

This tool is almost transparent to the programmer. The programmer just has to mark in a special way messages that need to be translated. It is also transparent to the translator since the precise location of messages in the source code and the relocation of messages. Translators just have to keep the translation of a list of messages current. The gettext tools handle the creation of catalogues and their regeneration when the sources change but, at the same time, preserving translations already done.

This way, the work of translating messages within a program is reduced to the task of doing an initial translation of all the messages and then the maintenance of the small (or big) changes introduced in the code that might have as a consequence the apparition (or change) of messages. The programmer has just to prepare the sources to use gettext through the use of the tools both in the program itself and in the tools that compile and build it (a process known as software internationalisation). Once this is done, the work of both groups can be done independently, which helps development both ways. That is, a translator does not have to depend on the programmer to introduce a new language and a programmer does not have to wait for the translators to do their job in order to publish a new release.

More information is available in the official gettext project pages[58]. If gettext is already installed, online documentation is available that can be read executing **info gettext**.

Translation of messages using gettext is very simple. A translator just needs to pick up a PO file either partially translated or completely untranslated and fill in the missing "gaps". Having many people with many different capabilities willing to cooperate makes it possible to collaborate without the need of in depth knowledge of the translation mechanism. This is how translation teams born. Several front-end interfaces to gettext are available, such as **gtranslator**, **kbabel**, **poedit** or emacs'po-mode. One of consequences of the availability of the **gettext** tools is that it is not necessary to have programming knowledge to work on translation. The only requirements for working in translations are the knowledge of the original language, the language it will be translated to and the ability to edit .po files (with multiples tools being available to ease the task as described above).

[58] http://www.gnu.org/software/gettext/gettext.html

```
# Copyright (C) 1999 Free Software Foundation, Inc.
# Javier Fernndez-Sanguino Pea<jfs@computer.org>, 1999.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"POT-Creation-Date: 1999-06-21 14:21+0200\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: Javier Fernndez-Sanguino Pea <jfs@computer.org>\n"
"Language-Team: ES <ES@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: ENCODING\n"

#: hello.c:86
#, c-format
msgid "Usage: %s [-htvm] [--help] [--traditional] [--version] [--mail]\n"
msgstr "Modo de uso: %s [-htvm] [--help][--traditional] [--version] [--mail]\n"
#: hello.c:148
msgid "This is GNU Hello, THE greeting printing program.\n"
msgstr "Este es Hola de GNU, EL programa que imprime un saludo.\n"

#
msgid "Translate this"
msgstr ""
```

Figure 15.1.: PO file example

The GNU project supports different groups of internationalisation. These groups are coordinated by a person in charge of the translation team. There is no requirements, however, to provide constant dedication within a translation group. Translation can be a discrete effort. The existence of these groups, however, guarantees the revision of these discrete efforts from anonymous contributors (many time users). These groups are also in charge of developing glossaries to make translation of programs uniform and to maintain and update translations whenever new programs are published.

This work is coordinated through the *Free Translation Project* at `http://translation.sourceforge.net/`. The state of translations can be reviewed through a database of translations and translators at `http://translation.sourceforge.net/cgi-bin/registry.cgi?team=index`.

Translation headers: wml

This section will be completed in further releases of this paper

Po for everything (po4a)

po4a[59] (*PO for anything*) is a set of tools originally developed by Martin Quinson with the goal of easing translations and, more interestingly, maintenance of translations, through the use of gettext tools in areas where they were not expected, like documentation.

Tools in the po4a package are used to update PO files from the original files and generates translated files from these PO files.

The usual process is made of one initial step (convert translations to PO files) and two recurrent steps (update PO files and regenerate translated files).

Converting existing/translated documentation to PO Converting documentation files to PO is the job of the `po4a-gettextize` command. When starting a new translation, `po4a-gettextize` will extract the translatable strings from the documentation file and write a POT file from it. When translated files exist, strings are grabbed from them and collected in a new PO file.

[59] `http://po4a.alioth.debian.org/`

The `po4a-gettextize` tool only extracts the Nth string from the translated file and matches it to the Nth string of the original file in the created PO file. It verifies that the original document and the translated documents have the same structure, but can't detect the state of the translation.

Because some manual expertise by translators is then required, all extracted strings are marked as "fuzzy" by the `po4a-gettextize` process.

Converting manpages to PO

Converting existent manpages to PO files can be done using the **po4a-gettextize** tool:

```
$ po4a-gettextize -f man -m the_manpage -p foo.po
```

Or if an existing translation exists:

```
$ po4a-gettextize -f man -m the_manpage -p foo.po -l existing_translation
```

PO files can be converted back to manpages with:

```
$ po4a-translate -f man -m original_manpage -p PO_file -l translated_manpage
```

Upstream authors can use the following Makefile to translate manpages by placing it in a `po` subdirectory under the documentation directory (which holds the manpages in troff format).

Maintaining translated documentation with po4a The dataflow can be summarised as "master document --¿ PO files --¿ translations". Any changes to the master document will be reflected in the PO files, and all changes to the PO files (either manual or caused by the changes from the master document) will be reflected in translated documents.

Po4a tools allow defining a minimum translation ratio below which the translated document will not be generated anymore or will only contain the original strings. This avoids generating documents with too few translated strings but still allows the publication of complete translations. As a consequence, translated documents may still contain some strings in the original language. This tries to guarantee the accuracy of the translated document.

This behaviour is a big improvement for documents such as manual pages. Indeed, an argument often used for not using these translations is the lack of guarantee that the translations are in sync with the original man pages. The po4a tools lower this argument, the price being sometimes mixed man pages, though.

Recommended po4a organisation for software This section is mostly taken from the **po4a** documentation and describes the recommended organisation for software that use **po4a**.

A standardised architecture of the source tree will help the translation teams when they try to detect the POTs that need to be updated. As a consequence, the following architecture is recommended:

```
/
/doc/
/doc/en/
/doc/en/
/doc/po4a/
/doc/po4a/add_<ll>/
/doc/po4a/po4a.cfg
/doc/po4a/po/
/doc/po4a/po/<pkg>.pot
/doc/po4a/po/<ll>.po
/doc/<ll>/
```

Or, if you want to avoid a big POT and split it according to the packages, documents, formats, or subjects, you can use the following architecture:

```
/
/doc/
/doc/en/
```

Figure 15.2.: po4a Makefile example

```
PROJECT=project_name

# Sections of the manpages
SECS=1 2 3 4 5 6 7 8

PO4ATRANSLATE  = po4a-translate
PO4AUPDATEPO   = po4a-updatepo
PO4AGETTEXTIZE = po4a-gettextize

MAN1 = $(notdir $(basename $(wildcard ../*.1)))
MAN2 = $(notdir $(basename $(wildcard ../*.2)))
MAN3 = $(notdir $(basename $(wildcard ../*.3)))
MAN4 = $(notdir $(basename $(wildcard ../*.4)))
MAN5 = $(notdir $(basename $(wildcard ../*.5)))
MAN6 = $(notdir $(basename $(wildcard ../*.6)))
MAN7 = $(notdir $(basename $(wildcard ../*.7)))
MAN8 = $(notdir $(basename $(wildcard ../*.8)))

# List the languages the manpages are translated to
LANGS = fr
TRANS=$(foreach lang,$(LANGS),\
                  $(foreach sec,$(SECS),\
                            $(foreach man,$(MAN$(sec)),$(man).$(lang).$(sec))))

PARTIALPOT=$(foreach sec,$(SECS), \
           $(foreach man,$(MAN$(sec)),$(man).$(sec).partial.pot))

all: $(TRANS)

#### CREATION OF THE POT FILE ####

$(PROJECT).doc.pot: $(PARTIALPOT)
        msgcat --use-first $^ -o $@

%.partial.pot: ../%
        $(PO4AGETTEXTIZE) -f man -m $< -p $@

#### CREATION OF THE POT FILE ####

$(PROJECT).doc.pot: $(PARTIALPOT)
        msgcat --use-first $^ -o $@

%.partial.pot: ../%
        $(PO4AGETTEXTIZE) -f man -m $< -p $@

#### TAKE CARE OF THE PO FILES ####

%.doc.po: $(PROJECT).doc.pot
        @echo -n "Merging $(PROJECT).doc.pot and $@: "
        @msgmerge $@ $(PROJECT).doc.pot -o $@.new
# Typically all that changes was a date. I'd prefer not to cvs commit such
# changes, so detect and ignore them.
        @if diff -q -I'#:' -I'POT-Creation-Date:' \
        -I'PO-Revision-Date:' $@ $@.new >/dev/null; then \
          rm -f $@.new; \
          touch $@; \
        else \
          mv -f $@.new $@; \
        fi
        @msgfmt --statistics $@
```

```
/doc/en/
/doc/po4a/
/doc/po4a/add_<ll>/
/doc/po4a/<pkg1>/po4a.cfg
/doc/po4a/<pkg1>/po/
/doc/po4a/<pkg1>/po/<pkg1>.pot
/doc/po4a/<pkg1>/po/<ll>.po
/doc/<ll>/
```

It is important to avoid a build failure if a translation cannot be generated (the PO is too outdated, an addendum cannot be applied, ...). Wildcards should therefore be used or tests be included to check whether the files are generated in the 'install' or 'dist' rules.

When po4a is used upstream, it is recommended to run po4a in the 'dist' rule. This will update the POT and POs, and will generate the translated documents.

These translated documents can be distributed in the source archive if the maintainer doesn't want to add a build dependency on po4a. This requires adding an autoconf check on po4a and will allow updating the documentation if po4a is available locally. If po4a is not available, documents will be distributed without being synced with the original version, but the build process won't fail.

It is important to distribute the POT and POs in the source archive.

When po4a is used in a distribution such as Debian, one should ensure that the source of the package contains only up-to-date POT and POs. This means that po4a should be run in the 'clean' rule of the make process (typically in **debian/rules** for Debian packages.

```
clean:
	# Update the POT and POs
	cd <...>/po4a && po4a --no-translations --rm-backups <package>.cfg

build:
	# Generate the translations
	cd <...>/po4a && po4a --rm-backups <package>.cfg
```

Debconf-updatepo and po-debconf tools

Tools in the po-debconf package are mostly aimed for maintainers use. The central tool is the debconf-updatepo utility. This utility should be run each time a change happens in debconf templates, i.e. most often the files names *.templates in the debian/ directory of the package source tree.

debconf-updatepo looks for all files listed in the `debian/po/POTFILES.in` and search in these files for translatable strings. Instructions on how to set a debconf string as translatable are given in the po-debconf(7) man page. This can be summarised as "just put an underscore character before the field names".

All translatable strings are written by **debconf-updatepo** in a "template file" named `debian/po/templates.pot`. In the same time, all PO files that are present in `debian/po` are updated with regards of the new strings.

Changed strings are marked *fuzzy* (thus keeping the old translation) or *untranslated*. In such case, the old translation is kept as *obsolete* entries et the end of the PO files.

"fuzzy" strings translations are never used. When a debconf template includes more than one fuzzy string, the whole template will be shown untranslated to users.

Maintainers should run **debconf-updatepo** as soon as they change templates. Some i18n maintainers recommend running it in the package clean target to ensure that all PO files AND the POT file are up-to-date. There is nothing worse than a package with obsolete files. Some other maintainers rule against running **debconf-updatepo** in the clean target as this is likely to modify files in the debian/ directory.

Rebuilding the templates file that will be shipped with the package, as of `debian/<package>/DEBIAN/templates` is the job of **po2debconf**. This utility is auto-magically called by **dh_installdebconf** for developers that use debhelper tools. This is actually a very good argument for using debhelper tools for packages that use debconf.

The final templates file uses the encoding defined in `debian/po/output`. The default value is "utf8". It is highly recommended to use "utf8" in all cases. There is actually no good reason to use anything else. This will not affect the way the templates will be displayed, even in non UTF-8 environments.

The podebconf-report-po utility The **podebconf-report-po** utility is aimed to send notices about needed updates to all translators who have incomplete files (at least one fuzzy or untranslated string).

Using it before planning an upload of a package with modified templates is strongly recommended. Translators do not really like to discover changes and needed updates after a package has been uploaded.

When using this utility, maintainers should remember that many translation teams need time for their internal QA processes to take place. Leaving only a few days for translators to update their work, especially when important changes occurred, is nearly similar to a call for bad translations.

Documentation's doc-check

Translation updates for documentation are handled through a script called **doc-check**. Although the tool shares a common name, different documents use a different version (or incarnation) of the same tool (or concept). Basically, the translators need to add to each translated file a translation header that specifies which file revision was translated. So, if a document was translated based on the 1.12 revision in CVS, a translator would add this header:

```
<!-- CVS revision of original english document "1.12" -->
```

The specific header format used might vary between documents, for example, this is the translation header for the translation of a file in the Debian Installation Guide:

```
<!-- original version: 12756 -->
```

If these headers are in place a translator can use the **doc-check** tool to:

- show which translated files have a different revision number from the original English files
- show what changes have been made in the original English file
- help keep track of which files have not yet been translated

In order to use the tool the translator just needs to update his/her local copy of the document source code, run the script and review the output. The use of a revision control system helps the translator extract[60] which changes have been introduced in the translated file. (S)he can then update the file, update the translation header and commit his/her changes.

This mechanism is indeed very similar (if not the same) to the mechanism used in the website to manage translations.

Translations that feel more comfortable using PO files can use either po4a (see Section 15), or **poxml**. The Debian Installer uses **poxml** and provides tools to convert from the XML files to PO files for the translation teams interested. A different set of tools converts the PO files back into XML files. "Out of dateness" of translations is handled through the same PO mechanisms (*fuzzy*) as those used by **gettext** to determine whether to show a translated message or not.

Use of the BTS for translation work

Translators As always, using Debian Bug Tracking System to deal with translation work and particularly to send translation updates or new translations, is the recommended way.

Translators are encourage to send new translations or translation updates as bug reports against the relevant package. The use of the **reportbug** utility is encouraged for translators working on Debian systems or Debian derivative systems (such as Debian-Edu or Ubuntu). The **reportbug** is part of the **reportbug** package which can be installed with the following command line:

```
aptitude install reportbug
```

This should be done as root or, if the current user is listed in *sudoers* by using the **sudo** command (this is useful on Ubuntu systems where the root user is not activated by default):

```
sudo aptitude install reportbug
```

Bug should be reported against the *source* package which the translation belongs to, preferably.

reportbug will prompt for a bug *title*. By general agreement, semi-formalised bug titles should be sued:

```
[INTL:<code>]: <language> <type> translation
```

[60] The **doc-check** script of the *d-i* project can extract the changes between revisions by making the proper queries to the revision control system if asked to.

where *¡code¿* is the language ISO-639 code, *¡language¿* is the language name in English and *¡type¿* is the type of translation (which can be ”po-debdonf”, ”programs”, ”man pages”, etc.)

reportbug will request for the bug severity to use. By general agreement in the project, the *wishlist* severity should be used.

reportbug will also prompt for *bug tags* which is a common way to categorise bug reports. The *l10n* and *patch* bug tags should be used.

reportbug then drops the user into a text editor to fill in the bug text. There is usually no need to be very verbose there as most informations are obvious.

Finally, the translation file should be attached to the bug report (**reportbug** is clever enough to warn users when a bug tagged *patch* is sent without any file attached. The attached translation file can be sent as is but several translators commonly compress these files with **gzip** to avoid maintainers mail user agent messing up with the file encoding.

A recent announcement[61] by Denis Barbier (FIXME: URL) proposed to use bugs usertagging to categorise l10n bugs.

This proposal sets up user tags such as *l-¡code;¿* where ¡code;¿ is the language code, *l-* being a prefix for *language* while *t-* could be a prefix for *territory* (or country). The *user* to use is *debian-i18n@lists.debian.org*. This proposal is currently not completely finalised and could change in the future.

Package maintainers Maintainers should use the sent files as is. They should NOT change them and, for instance NOT re-encode them. It is recommended that maintainers check that the translation is sent by someone who is either the former translator or someone with his/her approval. When in doubt, advice should be requested in the debian-i18n mailing list.

When maintainers receive new translations or translation updates for an upstream program, they must send them to upstream and possibly setup a good working method with their upstream for this translation updates flow. The translations of software in Debian should idealistically not be different from upstream translations.

Translation updates are registered exceptions to freezes that occur at the end of the Debian release cycle, so maintainers should make use of that general freeze time to make sure they have complete translations and possibly request for updates to their translators.

Relationship with other projects

This section will be completed in further releases of this paper

Translation packaging

This section will be completed in further releases of this paper

Handling of bug reports for upstream translations

The situation varies considerably between upstream projects which have a strong and well organised upstream i18n/l10n team, such as OpenOffice.org, KDE, Gnome and similar projects, or even ”smaller” projects where translations are handled through the Translation Project, and other projects for which the localisation is mostly done by direct interaction between the upstream maintainer and a variable set of volunteers.

For the first kind of project, it is recommended that Debian maintainers and translators avoid interfering with the upstream i18n/l10n resources. This means that translation updates coming through the Debian BTS should be redirected to the upstream i18n team, or to the Translation Project. In short, sending translation updates through the Debian BTS is here discouraged.

For projects where internationalisation and localisation is done autonomously, Debian maintainers can interact more closely with upstream. Most upstream maintainers may be deeply interested by the big amount of Debian resources for localisation and it is recommended to push for Debian translators to maintain as many upstream translations as possible, for such projects.

This needs a closer interaction with the upstream maintainers, for instance by requesting access to his/her revision control system or have him/her deal directly with the Debian BTS (tagging and triaging bugs). Such method has already proven to be a good method to develop a general stronger interaction with upstream, not only for localisation.

Handling of errors in upstream translations

This section will be completed in further releases of this paper

[61] `http://lists.debian.org/debian-devel-announce/2005/09/msg00002.html`

16. GPLv3 and Debian

by Don Armstrong

For the remainder of the year, we have an opportunity to participate in a historic process that affects a large part of the software in Debian, as well as the greater Free Software community. During this calendar year, the FSF is working on drafting a revision of the GNU Public License, and you all are an important part of that process. The current draft has changed many things, and it's quite likely that the subsequent drafts will contain more changes that affect the way that works under the GPLv3 are able to be used.

We will attempt in this paper to discuss the process behind the adoption of a GPLv3, that is, how changes to the current draft are going to be suggested, commented upon, stewarded, and made, the changes and the rationale behind the changes that have been made in the current draft of the GPLv3 and the GPLv2, and finally a non-exhaustive look at some of the major issues which have been identified with the current draft.

Drafting and Community Review Process

The drafting and review process has three important components which are going to interact together to result in subsequent drafts and eventually the final revision of the GPL version 3. These three components are the community at large, four committees made up of community members, and Richard M. Stallman (with the assistance of Eben Moglen and others at the SFLC.)

Community

The Free Software community (that includes you if you're reading this) is responsible for reviewing the various drafts of the GPL and making comments about the changes that have been made to the GPL (or even comments about things that haven't been changed, but possibly should have been.) The comments are made currently using a web based application at gplv3.fsf.org, and at the time I'm writing this a few thousand comments have been made. (In fact, almost no substantive portion of the license has not been commented upon.)

Hopefully, the community will be able to identify most of the serious issues with each of the revisions of the license, and begin to identify ways that the license can be changed (if necessary) or at the very least identify areas that need change or clarification. Since the community includes not only Free Software developers like yourselves, but attorneys who are either developers or users in their own right spread throughout many of the legal systems on the planet, it's quite likely that at least the obvious issues will be discovered and fixed. (Ideally by the sheer application of many eyeballs even the less obvious issues will be noticed and addressed.)

Committee

Five committees, four of which were organized at the first GPLv3 conference at MIT in January have been formed. Branden Robinson, Greg Pomerantz, Mako Hill, and Don Armstrong are project members who are also on these committees. Each of the committees was free to organize itself as it wished, although representatives from the FSF acted as guides throughout the process to keep the committees on track and on task.

The committee's primary responsibility is to examine the comments that are made by the community, group the comments into issues, and then make recommendations based upon the comments to resolve the issues that have been raised. For example, members of Committee D have been examining all of the comments that have been made on the GPL and assigning them to different categories based on personal assessment. Groups that are made by each member are then brought forward at the weekly meetings in IRC (#committeed on irc.freenode.net at 22:00 UTC on Tuesday) and if the members agree, are made issues, and someone volunteers to steward the issue.

Once an issue is identified, discussion on the issue occurs in committee, and if necessary, more comments/suggestions are made in the commenting system which are included. If changes are required to the GPL to resolve the issue, those changes are also made and discussed. Finally, assuming the members of the committee agree, a final position paper can be made, and the committee can submit the issue for consideration by RMS.

RMS

Richard M. Stallman is serving as the final arbitrator of changes that are made to the GPL. The committees will be presenting cases for any changes that need to be made to the license, and RMS will be weighing them, and making a final determination on them. RMS will of course be working in close concert with the FSF's legal representation, so Eben Moglen and the SFLC will also play a role in this part of the process.

In the course of the next year, RMS will be involved in publishing two to three drafts of the GPL, culminating at the end of the calendar year in a new version of the GPL, one that the Free Software community in general is content with and willing to license its works under.

Overview of the Current Draft

The current draft of the GPLv3 changes quite a few things from version 2. We'll discuss some of the more obvious changes here and retread the FSF's rationale for each of the changes that we discuss. We will withhold discussion of the issues that these changes have raised (and other pre-existing issues) for section **??**.

Digital Restrictions Management

The current draft of the GPLv3 includes a clause (§1 ¶2) which resolves a possible loophole with the GPL version 2, whereby a distributor could distribute the source code to a program, but restrict the ability of anyone else to make modifications to the program by the means of cryptographic keys or similar to either lock the hardware, or block interoperability between programs. Additionally, it was possible to restrict the output of a work to being unlocked using a specific set of keys (or a specific vetted viewer which was closed source.) This clause requires that these keys be provided as part of the Complete Corresponding Source Code .

In addition to specifically requiring certain keys, the current draft of the GPLv3 indicates that the license should be interpreted in a manner that disallows attempts to restrict users' freedom.(§3 ¶1) This clause also disallows the distribution of "covered works that illegally invade users' privacy"[1]

The current draft also indicates (§3 ¶2) that covered works do not constitute part of an effective technological protection method in an attempt to obviate DMCA protection for GPLed works.

License Compatibility

A whole set of clauses were added to the current draft of the GPLv3 to attempt to increase the compatibility between the GPL and other free software licenses. The first three of these clauses primarily deals with explicit compatibility with BSD-like licenses (§7 ¶4–6); as the GPL has long been considered to be compatible with the 3-clause BSD,[2] this just makes the compatibility explicit.

The next clause (§7 ¶7) makes the current draft of the GPLv3 compatible with the Affero Public License. In other words, it allows the license to require "functioning facilities that allow users to immediately obtain copies of its Complete Corresponding Source Code."

The final clause in this section allows for compatibility with licenses that impose a limited set of patent retaliation terms. (§7 ¶8) Briefly, it allows licenses to revoke permission for use of added parts wholly or partially upon the initiation of a software patent lawsuit, so long as the lawsuit is not retaliating against another software patent lawsuit and the lawsuit specifically targets part of the covered work.

Patent Licensing

Software patents are, unfortunatly, a serious problem that we people in many jurisdictions have been forced to deal with. The current draft of the GPLv3 deals with this in a few ways; first the traditional automatic licensing of downstream users survives, with the explicit allowal of patent retaliation restrictions, in all activities allowed by the license, including modified versions of the work. Secondly, the current draft of the GPLv3 requires that patent holders who are immune from suits under patent because they have a patent license must act to shield downstream users against possible patent infringment claims which their patent license protects them.(§11 ¶2)

[1] We'll discuss the problems with this particular clause in greater deal in section 16 on page 118

[2] That is to say the BSD license which does not have the advertising clause

Open Issues with the Current Draft

Digital Restrictions Management

The current wording of the sections on Digital Restrictions Management has caused a bit of consternation in many camps, including Linus Torvald's rather tepid reception of v3. First and foremost, it appears that the GPLv3 as written requires distributors to provide the keys used to sign a work, just like we do in Debian in order to help users assure that the code that they are installing is actualy the code that we have distributed to them. There are ways to interpret the language as written to not require this, but they are not as straightforward as one would like. Additionally, it appears to require that API keys which are used to uniquely identify the user of a work for billing or bandwidth/resource limitation purposes be provided along with the work. These were most likely not intended, and are merely an artifact of the very complex nature of trying to preserve user's freedom, while granting them the ability to do everything that they wish to do.

Secondly, the current clauses on DRM as written disallow the illegal invasion of user's privacy. This appears to have been included as a response to the Sony DRM debcale, and a desire to be able to go after people who illegally violate others privacy with a larger stick than the criminal laws in various jurisdictions currently afford. Unfortunatly, restricting users from performing specific sets of actions almost definetly contravenes DFSG §6[3] and more importantly, it doesn't do anything to solve the more serious problem of invasion of user's privacy where it is the government itself that is doing the invasion. Hopefully this clause, while well meaning, will be removed from the second draft of the GPLv3.

A final issue with this section comes from the anti-DMCA clause of the current draft of the GPLv3 (§3 ¶2). This clause has caused a bit of confusion in various people not familiar with the DMCA and the provisions provided for within. Additionally, it's not clear whether the clause will actually be able to keep a covered work from activating that clause of the DMCA, as very little case law exists in this area. Of course, at worst case, this clause will just become a confusing null-op, in the optimal case it may actually do some good.

Affero Clause

A more contentious clause is in §7d, the Affero compatibility clause. Briefly, the Affero compatibility clause is an attempt to close the ASP loophole; the ASP loophole being the ability of application service providers to use a GPLed work and make modifications to it without providing the source code to the modifications to their users and/or the community in general.

The clause does this by implementing a restriction on modification by requiring the work to maintain a facility to allow users to immediately download the complete corresponding source code to the work. Obstensibly, such a restriction on modification fails the most expansive readings of DFSG §3; and likewise has problems with Freedom Zero, the freedom to study a program and make modifications to it for any purpose. It also causes problems where a bug in this mechanism could place you in violation of the license, and precludes using code with this restriction in areas where the immediate download of Complete Corresponding Source Code is impossible.

That being said, the use of ASPs to lock users out of being able to modify their own software is a serious problem that we are going to have to resolve in one way or another, either through software licensing like this, or by users not accepting that computing model.

Patents

Software patents are, as those of you have been following the machinations of the EU in this issue know, the night stalker of the entire software world. Basically, if you're not a big player with a few thousand software patents, or an entity whose entire purpose is to litigate patents and not actually make software you're going to be affected by software patents some day. Already there is software in Debian which is likely DFSG free, but we are not able to distribute because of the likelihood of becoming the subject of patent lawsuits. In an attempt to combat this, the GPL has always had a clause which caused patent licenses to flow from the licensee to the licenseor. The addition of the patent shield now forces larger companies with diverse patent portfolios and cross licensing agreements to act to shield downstream users from the possibility of patent lawsuits when their licensing agreements protect them.

GPL as Free Software

A long time issue is the fact that the GPL itself cannot be modified. The reasons behind disallowing this modification typically boil down to a desire to restrict license proliferation. However, given the interpretation by the FSF that the only part of the license that must be removed from the license if the license is modified is the preamble, it seems that any license proliferation prohibition has been virtually eliminated.

[3] "The license must not restrict anyone from making use of the program in a specific field of endeavor."

Given the sheer number of people who are working on the GPLv3 and the number of issues that have been raised, it seems that no one should take writing a software license or modifying an existing one lightly. In that light, the lack of modifiability to the GPL has not been altogether too important.

However, it seems reasonable that we should apply the same philosophy to as many of the works as possible that we, as members of the Free Software community, create. Not that someone should modify the GPL, but just so we are philosophically consistent when we request others to join our community.

Conclusion

We're now 5 months into the process to draft the GPLv3; that means we've got less than 7 months left to suggest any changes to the new version of the draft that we can. If you haven't yet, you should take some time and read over the current draft of the GPLv3 , and make sure that it protects the freedoms that you feel are important to have as a member of the Free Software Community. Don't take our word for what the changes, problems, and rationale are for the current draft of the GPLv3 ; think about them, and comment on them too!

Part IV.

Appendix

A. (incomplete) List of Birth of the Feather Sessions

A.1. technical track

A.1.1. An Open Source Driver Framework

by Max Alt, Intel Corporation and Dario Rapisardi, Junta de Extremadura

Abstract

The discussion will center on two common hardware support issues that affect Debian and Debian-derived distributions: how to easily install and configure drivers for a given set of hardware and how to ensure hardware devices will function correctly once that software is installed. The presentation will survey questions and issues that Intel's customers have faced when deploying an ever-growing number of Debian derived distributions and will present a number of interesting analogies related to hardware support. This will segue into a discussion about an open source prototype application named "Topper", which is designed to maintain key knowledge on hardware and software dependencies. The application definition, scope, design and goals will be discussed along with Topper user interface and interface to validation environment. At the conclusion the speakers will demonstrate the prototype, in which a Q & A with the audience will be offered.

A.1.2. Making Music with Debian

by Paul Brossier

Abstract

This session will be focused on making music with our favorite operating system. On the menu:

- presentation of the major audio softwares
- Demudi, a Custom Debian Distribution
- status of audio software in the debian archives
- support for audio hardwares in the linux kernel
- more...

A.1.3. Alternative Developer's Interface to APT: libapt-front

by Petr Rockai

Abstract

This paper describes libapt-front, a new API for accessing APT. The library is designed to be both easy to use and powerful. While its primary target language is C++, we support binding efforts for other languages, namely python and ruby (and eventually others). The main part of the paper is concerned with overall library organisation and philosophy. It also contains a short summary on our development methodology, few usage examples and our plans for the future.

After the talk, interested parties are invited to a small workshop where we will demonstrate how to write a libapt-front based application from the ground up.

Eventually, we will be happy to hold a BoF about the library, mainly for people who would be interested to join our effort, to discuss the current status, plans for the future and so on.

A.1.4. Webapps Common: The Central Point in Developing a next-generation Web Server and Web Application Policy

by Neil McGovern

Abstract

The webapps policy aims to clarify the standards and technical requirements for web-based software in Debian. It also serves as packaging manual with examples of best practices for packaging web applications in Debian.

There is a draft policy which we're working on at our alioth project[1] via the mailing list[2].

I would like to take this opportunity to get everyone who's involved around a table and hammer out some of the points we've come across.

A.1.5. GNOME Maintainership in Debian

by Josselin Mouette, Jordi Mallach and Gustavo Noronha

Abstract

The GNOME packaging team has been in charge of a large part of GNOME packages in Debian since the end of 2003, and is now handling almost all packages of the GNOME desktop platform.

In this talk, we will present the way the team works, and the tools we are using to work together : subversion, IRC, cdbs, gnome-pkg-tools.

Then, some maintainers will show the Debian specifities of some packages, subsystems or branding, and how to integrate well with them. This includes hints about building a GNOME-based package, tricks for children distributions, and how to customize GNOME for a whole site.

A.1.6. Ubiquitous Cluster Computer with Debian

by Sritrusta Sukaridhoto

Abstract

Today, embedded processors (CPUs) can be found in a vast variety of products ranging from cellular phones, digital cameras and automobile navigation systems up to network-connected household appliances. Some of these embedded CPUs can run advanced operating systems, such as Linux, to achieve flexible network connectivity and to have logically same functionality as that of high-end CPUs designed for PCs and workstations. This trend accelerates the technology toward the age of *ubiquitous computing*, that is to integrate computation into the environment enabling people to interact with computers more naturally.

In such situation, distributed parallel processing with network-connected embedded CPUs will become one of the most important technologies to realize a variety of pervasive applications. One of the problems in managing R&D projects for ubiquitous/pervasive computing is the lack of cost-effective standardized platform for prototyping, testing and evaluating application programs on network-connected embedded CPUs. Addressing this problem, in this paper, we present a compact cluster computer with embedded CPUs, called a *Ubiquitous Computing Cluster (UCC)*, which provides a rapid-prototyping environment for ubiquitous/pervasive computing applications at very low cost compared with conventional PC clusters.

see more information at `http://www.aoki.ecei.tohoku.ac.jp/ucc/`

A.1.7. Common Lisp Development in Debian

by Peter Van Eynde

Abstract

Debian has several Common Lisp implementations and a variety of libraries. First we will explain the basic building blocks (implementations, libraries, slime and common-lisp-controller). We then show how to use the available libraries and then how to integrate your libraries in the Debian framework. Finally we demonstrate how to package simple Common Lisp packages using dh-lisp.

[1] `http://webapps-common.alioth.debian.org/draft/html/`

[2] `debian-webapps@lists.debian.org`

A.1.8. Debian and the $100 Laptop
by Benjamin Mako Hill

Abstract

The One Laptop Per Child project (better known at the $100 Laptop Project) has gained a huge amount of press. The goal of the project it provide hundreds of millions students across the world with affordable, durable computers through their schools and their country's ministries of education. The laptop, will be running Free Software.

While many people, including Red Hat, are working the project, this does not preclude the involvement of Debian developers or bar Debian from running on the machine. This BOF will aim to answer questions about the project an brainstorm ways that Debian developers, and Debian as a whole, can make an important impact on the project and on the world.

A.1.9. Indic & Debian
by Karunakar Guntupalli

Abstract

Indian language localization of Free and OpenSource is been happening for a while (6yrs), with many languages being fully supported. Debian and its derivatives have been the distribution of choice by localizers.

The IndLinux project has been on the fore front of Indic localization and over the last 3 years has been using debian as the distro for localization development.

This talk will highlight the developments in Indian language localization, problems faced, efforts to have Debian support all Indic languages and activities to promote Debian in India.

A.1.10. Scratchbox 2: Bringing Crosscompiling to Debian
by Riku Voipio

Abstract

Scratchbox is a mature toolkit which allows fast cross-compilation of unmodified sources - source configuration scripts wont even notice that they were crosscompiled.

However, Scratchbox binary debs take over 200MB combined. This space goes mostly to host tools (make,Perl,Python,flex,texinfo, and so on), compiled specifically for Scratchbox. Using native binaries wherever possible is important, as often most of Debian package building goes elsewhere than actual GCC calls.

Scratchbox 2 will no longer require host tools or crosscompiler toolchains to be compiled specifically for scratchbox. This will make the core of Scratchbox a small package possible to be uploaded to Debian.

With a clever open() / exec*() / etc redirecting LD_PRELOAD library and ld.so magic, Scratchbox 2.0 provides an environment where unmodified host and target binaries co-exist in different directories, and the LD_PRELOAD library selects automatically the correct tool to execute.

Plans include a "pbuilder create" style interface to create crosscompiling environments out of standard Sarge / Etch / Sid Debian packages with crosscompilers built out of Debian's toolchain packages. This makes it possible to use scratchbox in buildd's, at least for unofficial ports.

This talk will present the performance advantages of Scratchbox approach when building for low-end architectures. Scratchbox will be compared against approaches like native compilation, distcc callbacks to a crosscompiler or native compiler running in an emulator.

The separation of host tools and target libs allows easier bootstrapping of a new architecture, how about for example Debian on Cris/uClibc? Using crosscompiling and host tools most circular build dependencies can be beaten, and the rest build dependencies can topologically be sorted for a correct bootstrapping order.

A.1.11. Multithreading: Why and How we should use it
by Ben Hutchings

Abstract

Many programmers have avoided use of multiple threads, because they can be hard to use correctly and are often unnecessary on single-processor systems. Others have started using them without a full understanding of how the memory model of a multithreaded program can differ from that of a single-threaded program. For the foreseeable future, increases in computing power will largely be due to increased parallelism (multiple processor cores and multiple threads scheduled per core) and not by increases in the speed at which individual threads are executed. Therefore it is important

that programmers learn how to write reliable multithreaded code. The portability of Debian raises particular challenges, as different architectures implement different memory models in multiprocessor configurations.

A.1.12. Optimizing Boot Time
by Margarita Manterola

Abstract

The amount it takes for a common debian-installed machine to boot is incredibly long. Some of the problems can be quite easily fixed, some other require more work, but all of that can be corrected if we have that target in mind.

This informal talk aims at pointing the critical problems in boot time, proposing some solutions to them and discussing other solutions with the audience.

A.1.13. The ARM port and new ABI Transition
by Wookey

Abstract

Discussion of the state of the existing arm port and the proposed new ABI for ARM.

We'll cover the arm port's status as an underused, yet very important port, recent problems, new variants and future challenges.

There is strong pressure to move to a new ABI for ARM. This session will cover the practical effects this has on toolchain, kernel, libraries and applications. Pros and cons of Debian using the new ABI (for big and/or little-endian arm). How such a transition can be best managed within the Debian infrastructure, and considering the interaction between Debian and the commercial interest of ARM Corp.

A.1.14. Embedded Debian
by Wookey

Abstract

Embedded Debian has just had an Extremadura work session, SLIND has been released, cross-toolchains for all suites are available and Embedded Debian is now a reality. Come along to find out what works now and discuss future activities. If you don't know why you should care about Embedded come along. If you have a small system you want to run Debian on, come along. We need developers and we need to know what people want out of the project in order to direct resources.

A.2. social track

A.2.1. The debconf video team, the video archive and YOU
by Holger Levsen

Abstract

This roundtable or workshop serves to aims: First, to report how the video team has worked so far and get some feedback on this. Second, we want to give people, who just became involved, a complete picture of what the video team has been and will be doing in Oaxtepec to produce live streams and files, which will end up in `http://meetings-archive.debian.net/pub/debian-meetings/2006/debconf6/`

The debconf videoteam wants to motivate *YOU* to join us here and now (that is before this BOF will be held), so you can help us to improve our workflow, produce even better recordings and have a good time at debconf6 with a lighter workload then at debconf5.

For specific tasks, like operating the cameras or audio equiqment, we also plan to hold training sessions, which will take place just before DebianDay. Please don't wait for this BOF session if you want to become involved, show up and participate as early as you can!

The work of the video team is divided into the following areas:

- camera team
- audio mixer

- dv grabbing, streaming, encoding
- uploading, archiving
- collecting slides
- producing DVDs

Open questions:

- Scope:
  - is the archive for debian meetings only? i.e. from FOSDEM or LinuxTag only the debian specific talks were considered, should we try to have other free software content in the archive?
- meetings-archive.debian.net:
  - howto make this archive more team-maintained?
  - howto inform about new content? create a rss feed ? discuss & announce-mailinglists?
- Harddrive space:
  - hosting of the sources sensibly possible? (12gb per hour of video)
  - backup of the complete archive
- Licence:
  - is a software licence appropriate? switch to a new licence?
  - howto attach the licence to the ”binaries”?
  - create a ”licence and other meta-data” intro and outro?
- Sponsorship:
  - find a sponsor for DV cams for video team members

A.2.2. OpenSolaris and Debian: Can we be friends?

by Simon Phipps and Alvaro Lopez Ortega

Abstract

People keep coming to me and saying how great a Debian distribution with an OpenSolaris kernel would be. I agree. Let’s talk through the subject and see what needs to be fixed so the two communities can become firm friends.

The session will probably be a panel-led audience discussion, although a ”goldfish bowl” is another possibility depending on the initial panel. Awaiting responses from a mix of invitees from Debian and OpenSoalris, from legal and civilian.

A.2.3. The AJ Market: Making Free Software Expensive

by Anthony Towns

Abstract

The topic of this talk is my ”AJ Market” project. It will cover five aspects of the project – my motivations in creating it, the ideas from economics that support it, how to get people to actually participate in it, a review of its successes and failures, and a discussion of what comes next.

The audience for the talk is both free software developers who might be interested in thinking about setting up a similar system for themselves to help finance their habit, and also users of free software who’d like to encourage development in areas they don’t think are currently receiving enough attention – or at least those users who’re willing to put their money where their heart is. Folks who think money will simply destroy the essential volunteer nature of free software development should also enjoy the talk and the follow-on discussion.

The actual contents of the talk will depend on what happens with the AJ Market over the months between now and the conference; if it fails spectacularly, the talk will cover the ideas leading up to it, what went wrong, with less emphasis on helping other people duplicate it; if it does well, it will be more of a ”HOWTO”, and room for discussion on what the next level might be and how to get there.

See `http://www.erisian.com.au/market/` for more information.

A.2.4. Debian and Science

by Michael Banck

Abstract

As a community project with a diverse developer body, Debian has always been the distribution which included the most variety of scientific software not being limited by business constraints or a small developer community. While Computer Science, Mathematics and Physics (from now on referenced as ”the old three”) are traditional fields of endeavours for Debian Developers and users and therefore have seen attention in terms of packaging available Free Software applications since the beginning of the project, other fields of science have been taken care of by only a few individuals until some time ago, e.g. Andreas Tille for debian-med’s medical, biological and biomedical applications and Michael Banck for chemistry related applications. Along the same line, researchers and scientists from these fields seemed to be less pronounced among GNU/Linux users than for example Physicists, as the majority of them seems to be coming rather from a Windows background as opposed to the Unix background Physicists or Computer Scientists experienced. As GNU/Linux on the desktop matures and appears on the radar of regular users, scientists from other fields start to use it as well. Some of these will hopefully start helping out and become developers themselves.

Both the installation of the debian-science mailing list in summer 2005 and the rising popularity of the Debian-derived and desktop-centric Ubuntu distribution have increased the participation to Debian development of scientific packages outside of ”the old three” fields significantly in recent months. In order to channel these interests and avoid fragmentation between e.g. the Debian and Ubuntu communities, group maintenance should be encouraged wherever possible, as well as taking advance of other facilities developed over the last years:

- Group Maintenance through Alioth including the easy sponsoring of non-DDs
- The advent and success of easy to use Desktops such as GNOME and KDE
- Custom Debian Distribution frameworks make it simple to release specialised versions of Debian for various fields of science

The issues for group maintenance of packages from a specific scientific field are slightly different to conventional types of group maintenance e.g. the Debian GNOME team faces. Different package will usually be quite different in their build systems and packaging needs, but work is more easily split up by different people as packages do not tend to depend on each other that much usually. As most scientists in fields other than ”the old three” are usually not extensively trained and teached in programming, Free Software in these areas appears to be more bumpy both in terms of installation, maintainability and usability. Regarding installation, a lot of programs ship a simple Makefile at most and do not consider ever to be distributed as part of an operating system. Some rely on environmental variables for correct execution as well. Clearly, the need for improvement can be easily seen and should attract interested volunteers for low hanging fruits.

A big issue Debian has come across over the years is the popular use of ”almost Free” licenses for scientific applications. In the spirit of sharing research results, a log of scientists have been sharing their source code since the beginning, however, a lot do so with ”Academic use only” clauses. The mandatory citation in a scientific publication should a program have been used during the preparation of the research results is also a common clause.

This talk will review the current status of scientific software in Debian, with special focus on the upcoming Debichem sub-project and its experience in cooperating with the Ubuntu MOTUScience project.

A.2.5. Representing Debian - Doing the best for the best?

by Alexander Schmehl

Abstract

More and more often Debian is getting invited to different exhibitions, tradeshows and other similar events all over the world. Often Debian must decline such nice invitations, since we lack volunteers to staff the booths.

At this point Debian fails to motivate enough people. It think, this could be a problem of missing documentation on how to organize and set up a sucessfull Debian booth.

In this session I would like to present a list of basic hints, on how things can be organised, based on the experiences I made during the last years. I’m very interessted in hearing other peoples opinions and experiences on common problems, e.g.: how to advertise Debian’s presence at an event (people visiting our booths are often suprised to find Debian there and found it by pure accident), how to obtain advertising and merchandising material (CDs or DVDs as give aways, flyers, t-shirts, lanyards, etc), what to present there, perhaps even some does and donts for behaviour and how to deal with ”difficult” visitors.

I hope that after this session we might be able to improve and update existing howtos and documentation, which will hopefully motivate more people to help Debian by representing Debian.

A.2.6. What Society can learn from the Debian Project Experiment!
by Emmanuel Galatoulass

Abstract

In my talk I would like to point out the defining organisational aspects in the structure of the Debian Project and its activities, explicating the way they shape democracy in its functioning and argue that they can make a valuable lesson for broader social contexts, being an inspiring and thought-provoking model in implementing collaboration and democratic operation on a large scale. I am going to use as study cases the process of the DPL election and other Debian decision making procedures.

A.2.7. How Communication Works in Debian (Or what did I say to this Guy and why he is so upset with me)
by Jesus Climent

Abstract

The idea is to give an insight introduction on how communication works in debian, so people starting a collaboration with the project don't feel ignored, rejected, flamed, abused... and learn to deal with not so friendly situations (or otherwise too abusive attitudes).

A.2.8. recopilacion de uso de debian en mexico
by jesus christian cruz acono

Abstract

recopilacion de datos sobre debian y su uso y futura implementacion ademas de sus estadicas (incluye metadistros y derivados de debian

A.2.9. Ubuntu Annual Report
by Mark Shuttleworth

Abstract

Mark will report on the activities in Ubuntu since last year's Debconf, and respond to questions about the current progress being made in Ubuntu. Also, he will discuss plans for the next as-yet-unnamed version of Ubuntu, due for release in October 2006. This will be a good opportunity to address some of the threads that have raged on debian-devel abount Ubuntu directly with the project founder.

A.3. political track

A.3.1. Fourth Annual SPI@Debconf Workshop
by Benjamin Mako Hill

Abtract

Software in the Public Interest, Inc. is the non-profit charitable organization that holds Debian's assets, money, and trademarks and provides financial, legal, and other forms of support for the Debian project. It also supports a host of smaller free software and open source projects. During the last three years, the SPI workshop at Debconf has given Debian developers and SPI members a chance to voice criticism, comments, and feedback for SPI that has been taken to heart by SPI and put into action for the project. This year aims to build on and improve previous workshops.

A.3.2. Governance of the Debian Project
by Bdale Garbee

Abstract

The organizational structure of the Debian distribution has evolved a great deal since the project first started. This talk by a long-time contributor to Debian and former DPL will review key events in this evolution, and provide an insider's view on how the recent "DPL Team" experiment began and the impact it has had on the project. Finally, some ideas about how the Debian project can and should continue to evolve to meet ever-changing expectations will be presented.

A.3.3. Debian and the Law: Selected Legal Topics for the Developer

by Gregory Pomerantz

Abstract

As Debian grows and becomes more successful, keeping Debian on the right side of the law will become increasingly important. This talk will begin with a basic introduction to some of the legal considerations affecting free software developers and Debian in particular, including copyright, patent, trademark and trade secret law.

I will cover various legal tests for infringement of the various forms of ”intellectual property,” and propose some common sense strategies to help keep Debian out of trouble. There will be plenty of time for Q&A, so come armed with your law-related questions.

B. copyright notices

B.1. Internationalisation and localisation in Debian

B.2. Debian Community Guidelines

B.3. Nobody expects the Finnish inquisition

B.4. WTFM2: Avoiding Tears and Loss Of Hair

B.5. Security Enhanced Virtual Machines – An Introduction and Recipe

B.6. Debian’s Debugging Debacle – The Debrief

B.7. Other Documents

B.8. Licenses

On Debian systems the actual text of the license texts are available at `/usr/share/common-licenses/GPL` for the GNU General Public License and `/usr/share/common-licenses/Artistic` for The ”Artistic License”.
Printed copies of them are available at the organisation team’s room. Feel free to ask for them.

C. credits

It would have been impossible to make such a large event reality without the outstanding help of a great group of volunteers. Starting with those whom we couldn't name, since they just volunteered during the conference to do various task and ending by the core team, who started organizing this year's DebConf right after the last one ended.

It's impossible to list everyone who contributed to make this event come true; there are too many contributors. We are sorry for every name we forgot to mention here:

Core Organising Team

Main Organizer: Andreas Schuldei

Co-Organisers: Gunnar Wolf, Joerg "Ganneff" Jaspert, Neil "Maulkin" McGovern, Alexander "Tolimar" Schmehl and Holger Levsen

Sponsor Acquisition: Andreas Schuldei

Website Maintainer: Neil "Maulkin" McGovern

Graphical Artist: Agnieszka "pixelgirl" Czajkowska

Admin-master / BOFH: Joerg "Ganneff" Jaspert and Holger Levsen

Registration Coordinators: Gunnar Wolf and Rudy Godoy

Secretary: Marcela "Asciigirl" Tiznado and Luk Claes

Committee Members

Sponsorship Committee: Joerg Jaspert, Andreas Schuldei, Gunnar Wolf, Branden Robinson, Neil McGovern, Amaya Rodrigo Sastre, Margarita Manterola, Marcela Tiznado and Faidon Liambotis

Academic Committee: Joerg Jaspert, Andreas Schuldei, Alexander Schmehl and Meike Reichle

Local Organising Team

Mexican Organisation representing DebConf6: AMESOL, President: Jos Luis Chiquete Valdivieso

Main Local Organisers: Gunnar Wolf, Gabriela "Nadezhda" Manjarrez and David Moreno Garza

Accountant: Gabriela "Nadezhda" Manjarrez

Facilities Coordinator: Gabriela "Nadezhda" Manjarrez

Volunteer Manager: Marcela "Asciigirl" Tiznado

Speaker Team:

Talks Coordinators: Alexander "Tolimar" Schmehl and Don "dondelelcaro" Armstrong

Talks Moderators: Nattie Mayer-Hutchings and Alexander ”Tolimar” Schmehl

Speaker trainer: Meike ”alphascorpii” Reichle

Editors of the Proceedings: Alexander ”Tolimar” Schmehl and Gunnar Wolf

Video Team:

Video Team Coordinator: Holger Levsen

Editors: Herman Robak and Danny Cautaert

Camera Operators: Herman Robak, Danny Cautaert, Tore S. Bekkedal, Annabelle ”pixie” Tully, Nattie Mayer-Hutchings, Antonio Ognio, Neil McGovern, Ben Hutchings and Peter ”p2-mate” De Schrijver

Network Team:

Conference network-team coordinators: Holger Levsen and Joerg ”Ganneff” Jaspert

Network Team Helpers: Luciano Bello, Luk Claes and Alexander ”Tolimar” Schmehl

. . . special thanks to the volunteers working on all aspects of free software all over the world!
The Debian Project couldn't exists without you!

debconf sponsors

design by: creative_geeks - http://www.creative-geeks.org