6 Open Source Software Development: Future or Fad?

Srdjan Rusovan, Mark Lawford, and David Lorge Parnas

This chapter discusses the quality of Open Source Software (OSS). Questions about the quality of OSS were raised during our effort to apply experimental software inspection techniques (Parnas 1994b) to the ARP (Address Resolution Protocol) module in the Linux implementation of TCP/IP. It (1) reviews OSS development, (2) discusses that approach in the light of earlier observations about software development, (3) explains the role of ARP, (4) discusses problems that we observed in the Linux implementation of ARP, and (5) concludes with some tentative observations about OSS.

Ultimately, It's the Product that Counts

In recent decades, there have been so many problem software projects—projects that did not produce products, produced inadequate products, or produced products that were late or over budget—that researchers have become very concerned about the process by which organizations develop software. Many processes have been proposed—often with claims that they are a kind of panacea that will greatly reduce the number of problem projects. Process researchers are generally concerned with the "people side" of software development, looking at issues such as the organization of teams and project management.

We sometimes seem to lose sight of the fact that a software development process is just a means of producing a product and we should not ignore the quality of the product. We expect more from a real software product than that the current version "works." We expect it to have an "architecture"[1] that makes it practical to correct or update the product when changes are required.

This chapter reports on our look at one very well-known OSS product, the Linux operating system. What we learned by studying one component

of Linux raises some important issues about the process by which it was developed.

A Brief History of Linux

Linux was initially developed by Linus Torvalds in 1991. Linux has been revised many times since then. The work is done by a group of volunteers who communicate through the linux-kernel mailing list on the Internet. Torvalds has acted as the main kernel developer and exercised some control over the development. Commercial companies have added value by packaging the code and distributing it with documentation.

Linux is a Unix-like operating system. Most of the common Unix tools and programs can run under Linux and it includes most modern Unix features. Linux was initially developed for the Intel 80386 microprocessor. Over the years, developers have made Linux available on other architectures. Most of the platform-dependent code was moved into platform-specific modules that support a common interface.

Linux is a kernel; it does not include all the applications such as file system utilities, graphical desktops (including windowing systems), system administrator commands, text editors, compilers, and so forth. However, most of these programs are freely available under the GNU General Public License and can be installed in a file system supported by Linux (Bovet and Cesati 2000).

Introduction to Open Source Software (OSS)

Linux is one of the most widely available pieces of "open source" software; some people believe Linux and open source are synonymous. However, the "open source" concept has been applied to many other software products.

What Is Open Source Software?

In traditional commercial software development, software is treated as valuable intellectual property; the source code is not distributed and is protected by copyright and license agreements. Developers have gone to court to deny government agencies the right to inspect their software, and there have been lawsuits because a developer believed that source code had been stolen and used without permission.

In contrast, OSS is distributed with complete source code and recipients are encouraged to read the code and even to modify it to meet their indi-

vidual needs. Moreover, recipients are encouraged to make their changes available to other users, and many of their changes are incorporated into the source code that is distributed to all users. There are many varieties of OSS approaches, and many subtle issues about how to make them work, but the essence is to reject the assumption that source is private property that must be protected from outsiders.

The success of Linux and other open source products have demonstrated that OSS distribution is workable. Some products that were once proprietary have become open source and some products are available in both open source and proprietary versions.

"Brooks's Law" and Open Source Software

In his classic work *The Mythical Man Month*, Fred Brooks (1995) describes one of the fundamental facts about software development: adding more programmers to a project doesn't necessarily reduce time to completion; in fact, it can delay completion.

Intuitively, it might seem that adding programmers would increase the amount of programming that gets done (because two programmers can write more code than one programmer), but that does not mean that the goals of the project will necessarily be achieved sooner. A number of factors may contribute to the phenomenon that Brooks describes:

- Unless the coding assignments have been carefully chosen, the total amount of code written may increase as several programmers solve shared problems with code that is not shared.
- It is not often the case that two programmers can work without communicating with each other. Adding programmers often increases the number of interfaces between coding assignments (modules). Whether or not the interfaces are defined and documented, the interfaces exist and programmers must spend time studying them. If the interfaces are not accurately and completely documented, programmers will spend time consulting with other programmers or reading their code.
- Programmers must spend time implementing methods of communicating between modules.
- Often, some programmers duplicate some function already provided by others.
- It is often necessary to change interfaces, and when this happens, the programmers who are affected by the change must negotiate[2] a new interface. This process can seriously reduce the rate of progress. When two programmers are discussing their code, neither is writing more.

Brooks's observations should make us ask how open source software development could possibly succeed. One advantage of the open source approach is its ability to bring the effort of a worldwide legion of programmers to bear on a software development project, but Brooks's observations suggest that increasing the number of programmers might be counterproductive.

Two factors should be noted when considering Brooks's observations:

- Brooks's observations were about the development of new code, not the analysis or revision of existing code. We see no reason not to have several programmers review a program simultaneously. Of course, two people may agree that the code is wrong but identify different causes and propose different, perhaps inconsistent, changes. As soon as we start to consider changes, we are back in a code development situation and Brooks's observations are relevant.
- Brooks's observations are most relevant when the code structure does not have well-documented module interfaces. Since the original publication of Brooks's observations, many programmers, (not just open source programmers) have accepted the fact that software should be modular. If the code is organized as a set of modules with precisely documented stable interfaces, programmers can work independently of each other; this can ameliorate some of the problems that Brooks observed.

Open Source Software Is Not the Same as Free Software

One of the main advantages of Linux is the fact that it is "free software." It is important to understand that "free" means much more than "zero price."[3] "Free" is being used as in "free spirit," "free thought," and perhaps even "free love." The software is unfettered by traditional intellectual property restrictions.

More precisely, "free software" refers to the users' freedom to run, copy, distribute, study, change, and improve the software. In addition to the permission to download the source code without paying a fee, the literature identifies four freedoms, for the recipients of the software (Raymond 2001):

- The freedom to run the program, for any purpose.
- The freedom to study how the program works, and adapt it to one's own needs.
- The freedom to redistribute copies to others.
- The freedom to improve the program, and release improvements to the public, in the expectation that the whole community benefits from the changes.

A program is considered "free" software if users have all of these freedoms. These freedoms result (among other things) in one not having to request, or pay for, permission to use or alter the software. Users of such software are free to make modifications and use them privately in their own work; they need not even mention that such modifications exist.

It is important to see that the four freedoms are independent. Source code can be made available with limits on how it is used, restrictions on changes, or without the right to redistribute copies. One could even make the source available to everyone but demand payment each time that it is used (just as radio stations pay for playing a recording).

It is also important to note that "open source" does not mean "non-commercial." Many who develop and distribute Linux do so for commercial purposes. Even software that has the four freedoms may be made available by authors who earn money by giving courses, selling books or selling extended versions.

Open Source Development of Linux

The fact that all recipients are permitted to revise code does not mean that the project needs no organization. There is a clear structure for the Linux development process.

A significant part of the Linux kernel development is devoted to diagnosing bugs. At any given time, only one version (the "stable kernel") is considered debugged. There is another version of the kernel called the *development kernel*; it undergoes months of debugging after a feature freeze. This doesn't mean that the kernel is inherently buggy. On the contrary, the Linux kernel is a relatively mature and stable body of code. However, Linux is both complex and important. The complexity means that bugs in new code are to be expected; the importance means that a new version should not be released before finding and correcting those bugs.

The Linux development community has a hierarchical structure; small "clans" work on individual projects under the direction of a team leader who takes responsibility for integrating that clan's work with that of the rest of the Linux developers "tribe."

If one understands the Linux development process, the power of open source software development becomes apparent. Open source projects can attract a larger body of talented programmers than any one commercial project. However, the effective use of so many programmers requires that projects follow good coding practices, producing a modular design with well-defined interfaces, and allowing ample time for review, testing, and debugging.

By some standards, the Linux kernel is highly modular. The division into stable and development versions is intended to minimize interference between teams. At a lower level, the kernel has followed a strictly modular design, particularly with respect to the development of device drivers. Programmers working on USB support for version 2.4 of the Linux kernel have been able to work independently of those programmers who are working to support the latest networking cards for the same version. However, as we illustrate later, the interfaces between these modules are complex and poorly documented; the "separation of concerns" is not what it could be.

In the next section, we discuss the part of Linux that implements a part of the TCP/IP protocol to see how well the process really worked in that one case. We begin with a short tutorial on the protocol. It is included so that readers can appreciate the complexity of the task and understand how critical it is that the code be correct. The following description is only a sketch. It has been extracted from more detailed descriptions (Steven 1994; Comer 2000) for the convenience of the reader.

Communication Across the Internet

Programs that use physical communications networks to communicate over the internet must use TCP/IP (Transmission Control Protocol/Internet Protocol). Only if they adhere to this protocol are Internet applications interoperable. (Details on the conventions that constitute TCP/IP can be found in Steven 1994 and Comer 2000.)

The Internet is actually a collection of smaller networks. A subnetwork on the Internet can be a local area network like an Ethernet LAN, a wide area network, or a point-to-point link between two machines. TCP/IP must deal with any type of subnetwork.

Each host on the Internet is assigned a unique 32-bit Internet Protocol (IP) address. IP addresses do not actually denote a computer; they denote a connection path through the network. A computer may be removed and replaced by another without changing the IP address. However, if a host computer is moved from one subnetwork to another, its IP address must change.

Local network hardware uses physical addresses to communicate with an individual computer. The local network hardware functions without reference to the IP address and can usually function even if the subnetwork is not connected to the internet. Changes within a local network may result in a change in the physical address but not require a change in the IP address.

Address Resolution Protocol (ARP)

The Address Resolution Protocol (ARP) converts between physical addresses and IP addresses. ARP is a low-level protocol that hides the underlying physical addressing, permitting Internet applications to be written without any knowledge of the physical structure. ARP requires messages that travel across the network conveying address translation information, so that data is delivered to the right physical computer even though it was addressed using an IP address.

When ARP messages travel from one machine to another, they are carried in physical frames. The frame is made up of data link layer "packets." These packets contain address information that is required by the physical network software.

The ARP Cache To keep the number of ARP frames broadcast to a minimum, many TCP/IP protocol implementations incorporate an *ARP cache*, a table of recently resolved IP addresses and their corresponding physical addresses. The ARP cache is checked before sending an ARP request frame.

The sender's IP-to-physical address binding is included in every ARP broadcast: receivers update the IP-to-physical address binding information in their cache before producing an ARP packet.

The software that implements the ARP is divided into two parts: the first part maps an IP address to a physical address (this is done by the `arp_map` function in the Linux ARP module) when sending a packet, and the second part answers ARP requests from other machines.

Processing of ARP Messages ARP messages travel enclosed in a frame of a physical network, such as an Ethernet frame. Inside the frame, the packet is in the data portion. The sender places a code in the header of the frame to allow receiving machines to identify the frame as carrying an ARP message.

When the ARP software receives a destination IP address, it consults its ARP cache to see if it knows the mapping from the IP address to physical address. If it does, the ARP software extracts the physical address, places the data in frame using that address, and sends the frame (this is done by the `arp_send` function in the Linux ARP module). If it does not know the mapping, the software must broadcast an ARP request and wait for reply (this is done by `arp_set`, a predefined function in the Linux ARP module).

During a broadcast, the target machine may be temporarily malfunctioning or may be too busy to accept the request. If so, the sender might

not receive a reply or the reply might be delayed. During this time, the host must store the original outgoing packet so it can be sent once the address has been resolved. The host must decide whether to allow other application programs to proceed while it processes an ARP request (most do). If so, the ARP software must handle the case where an application generates an additional ARP request for the same address.

For example, if machine A has obtained a binding for machine B and subsequently B's hardware fails and is replaced, A may use a nonexistent hardware address. Therefore it is important to have ARP cache entries removed after some period.

When an ARP packet is received, the ARP software first extracts the sender's IP and hardware addresses. A check is made to determine whether a cache entry already exists for the sender. Should such a cache entry be found for the given IP address, the handler updates that entry by rewriting the physical address as obtained from the packet. The rest of the packet is than processed (this is done by the `arp_rcv` function in the Linux ARP module).

When an ARP request is received, the ARP software examines the target address to ascertain whether it is the intended recipient of the packet. If the packet is about a mapping to some other machine, it is ignored. Otherwise, the ARP software sends a reply to the sender by supplying its physical hardware address, and adds the sender's address pair to its cache (if it's not already present). This is done by the `arp_req_get` and `arp_req_set` functions in the Linux ARP module.

During the period between when a machine broadcasts its ARP request and when it receives a reply, additional requests for the same address may be generated. The ARP software must remember that a request has already been sent and not issue more.

Once a reply has been received and the address binding is known, the relevant packets are placed into a frame, using the address binding to fill the physical destination address. If the machine did not issue a request for an IP address in any reply received, the ARP software updates the sender's entry in its cache (this is done by the `arp_req_set` function in the Linux ARP module), then stops processing the packet.

ARP Packet Format ARP packets do not have a fixed format header. To make ARP useful for a variety of network technologies, the length of fields that contain addresses is dependent upon the type of network being used. To make it possible to interpret an arbitrary ARP message, the header includes fixed fields near the beginning that specify the lengths of the

addresses found in subsequent fields of the packet. The ARP message format is general enough to allow it to be used with a broad variety of physical addresses and all conceivable protocol addresses.

Proxy ARP Sometimes it is useful to have a device respond to ARP broadcasts on behalf of another device. This is particularly useful on networks with dial-in servers that connect remote users to the local network. A remote user might have an IP address that appears to be on the local network, but the user's system would not be reachable when a message is received, because it is actually connected intermittently through a dial-in server.

Systems that were trying to communicate with this node would not know whether the device was local, and would use ARP to try and find the associated hardware address. Since the system is remote, it does not respond to the ARP lookups; instead, a request is handled through Proxy ARP, which allows a dial-in server to respond to ARP broadcasts on behalf of any remote devices that it services.

Concurrency and Timing

In reading the previous sketch of the implementation of ARP, it must be remembered that this protocol is used for communication between computers on a network, and that many processes are active at the same time. There is concurrent activity on each computer and the computers involved are communicating concurrently. Opportunities for deadlocks and "race conditions" abound. Certain processes will time-out if the communication is too slow. Moreover, rapid completion of this communication is essential for acceptable performance in many applications.

Linux ARP Kernel Module

The Linux ARP kernel protocol module implements the Address Resolution Protocol. We have seen that this is a very complex task. The responsible module must perform the task precisely as specified, because it will be interacting with other computers that may be running different operating systems. One would expect this module to be especially well written and documented. This section reports on our review of this code.

Analysis of the ARP Module

Linux implements ARP in the source file *net/ipv4/arp.c*, which contains nineteen functions. They are `arp_mc_map`, `arp_constructor`, `arp_error_report`, `arp_solicit`, `arp_set_predefined`, `arp_find`,

`arp_bind_neighbour`, `arp_send`, `parp_redo`, `arp_rcv`, `arp_req_set`, `arp_state_to_flags`, `arp_req_get`, `arp_req_delete`, `arp_ioctl`, `arp_get_info`, `arp_ifdown`, `initfunc`, and `ax2asc`.

Linux ARP as a Module

We wanted to evaluate the ARP module[4] because it is a critical component of the operating system for most users, because it is inherently complex, and because it has to be correct. We expected to find a structure that allows modules to be designed, tested, and changed independently; that is, a structure in which you can modify the implementation of one module without looking at the internal design of others. This condition requires that the modules' interfaces[5] be well documented, easily understood, and designed so that it need not change if there are changes in its implementation or internal interfaces with hardware and other modules. Every well defined module should have an interface that provides the only means to access the services provided by the module.

We found the Linux networking code difficult to read. One problem was the use of function pointers. To understand the code and the dereferencing of a function pointer, it is necessary to determine when, where, and why the pointer was set. A few lines of comment directing people in this regard would have been incredibly helpful. Without them, one is required to search the full code in order to be able to understand portions of it. Such situations have negative implications for both reliability and security. Unless they have already become familiar with it, Linux TCP/IP code is difficult for even the most experienced programmers.

We found nothing that we could identify as a precise specification of the module and nothing that we consider to be good design documentation.[6] This is a serious fault. Even if one has read all the code and understands what it does, it is impossible to deduce from the code what the expected semantics of an interface are. We cannot deduce the requirements unless we assume that the code is 100% correct, will never change, and all of the properties of the code are not required by programs that interact with it. The inability to distinguish between required properties and incidental properties of the present code will make it difficult to write new versions of the kernel.

With properly documented interfaces, it would be possible to find bugs by confirming that the code on both sides of an interface obeys the documented semantics; a programmer would not need gness what each component was intended to do.

The Linux ARP module includes 31 different header files; most of them are long, ranging from a few hundred to a few thousand lines. It was very difficult to investigate all of them and find connections between every function in the ARP module and other functions inside and outside the module. Functions from the ARP module call functions from other modules. It is not a problem to find the functions that are directly invoked, but often those functions call some other functions in some other module. There are many indirect invocations resulting in many potential cycles. Some of those functions return values, most of which are not explained. We are not told what the returned values represent, and cannot even find some reasonable comment about them. We can only guess what they represent.

Many of the header files are implemented in other source modules. Since all calls to functions are interpreted using header files, it is impossible to understand and check the ARP source module without looking at the internals of other modules.

Design and Documentation Problems in the Linux ARP Module

Concrete Examples

The source file `neighbour.c`, in `net/core/neighbour.c`, includes 40 functions. Only ten of them are called by arp functions from the `arp.c module`. Those ten functions call many other functions. It is unreasonably hard to determine how those ten functions interact with the other thirty. These functions are:

- `neigh_if down` (this function is called by `arp_ifdown`)
- `neigh_lookup` (this function is called by `arp_find`)
- `pneigh_lookup` (this function is called by `arp_rcv`)
- `pneigh_delete` (this function is called by `arp_req_delete`)
- `neigh_update` (this function is called by `arp_rcv`)
- `neigh_event_ns` (this function is called by `arp_rcv`)
- `pneigh_enqueue` (this function is called by `arp_rcv`)
- `neigh_table_init` (this function is called by `init_func`)
- `neigh_app_ns` (this function is called by `arp_rcv`)
- `neigh_sysctl_register` (this function is called by `init_func`)

Just one `neighbour.c` function is called from the `arp.c` module: `neigh_ifdown (struct neigh_table *tbl, struct device*dev)` then calls the next functions: `atomic read (andtbl → lock), start_bh_atomic (), atomic_read (andn → refcnt), deltimer`

(andn → timer), neigh_destroy (n), deltimer (andtbl → proxy_queue).

None of these functions are explained or documented. All of them continue to call other functions without any kind of guidance to people who are trying to understand code. We found several books and papers about this code (for instance, Bovet and Cesati 2000), but none of them answer questions in detail.

The source file neighbour.c also includes 11 different header files but 7 of them (linux/config.h, linux/types.h, linux/kernel.h, linux/socket.h, linux/sched.h, linux/netdevice.h, and net/sock.h) are the same ones that are included in the arp.c source file. That makes the interface unnecessarily big and complex. The arp.c module (that is, Linux C networking code) is also complex; 19 ARP functions call 114 different functions outside of the module arp.c.

Some of arp.c functions—arp_set_predefined (13 different calls), arp_rcv (16 different calls), arp_ioctl (9 different calls), arp_get info (10 different calls)—are especially difficult for handling and understanding.

The file arp.h should declare the interface for the ARP module. It appears to declare eight access functions. However, it also includes two other header files, which then in turn include additional header files. A simplified version of the *includes hierarchy* resulting from arp.h is represented in figure 6.1. The actual *includes hierarchy* is more complicated, as 22 files that are included only from the file sched.h have been summarized as a single node in the graph in order to improve the figure's readability. Once the transitive closure of the includes is taken into account, file arp.h includes an additional 51 header files!

One of the two "includes" that appear explicitly in file arp.h declares the interface for the file net/neighbour.h, which contains the interface declarations and data structure from the net/core/neighbour.c code used by the ARP module. That one file contains approximately 36 function prototypes.[7] Many of the other header files not explicitly included in arp.h also contain additional function prototypes. In our view, this file illustrates a thoroughly unprofessional style of programming and documentation, violating the principles of information hiding by making all of the access functions and many of the data structures from lower-level modules implicitly available to any module using the ARP module.

Impressions

Our impression is that the whole hierarchy and relation between source modules (*.c) could be simpler. Lots of functions and header files are

Figure 6.1
Includes hierarchy for arp.h file

redundant and too repetitive. However, in code of this sort, without help of documentation, anyone would be afraid to make the changes.

Analyzing the ARP module felt like walking through a dark forest without a map. There were no directions explaining what given functions are really doing or clear explanations about their connections to another modules. It was not possible to understand most of the written code and it was not possible to define ARP as a module in the sense that we described at the beginning of this section.

Conclusions

Nobody should draw conclusions about a software development method by looking at one example. However, one example should raise questions and act as a warning.

The ARP is a critical module in any modern operating system and must conform precisely to a set of complex rules. Because it serves as an interface with other computers, it is likely that it will have to be changed when network standards are improved. It is reasonable to expect this module to be of the highest quality, well structured, well documented and as "lean" as possible.

Our examination of the Linux ARP code has revealed quite the opposite. The code is poorly documented, the interfaces are complex, and the module cannot be understood without first understanding, what should be internal details of other modules. Even the inline comments suggest that changes have been made without adequate testing and control. A cursory examination of the same module in operating systems that were developed by conventional (strictly controlled source code) methods did not show the same problems.

Nothing that we found in examining this code would suggest that the process that produced it should be used as a model for other projects. What we did find in this case is exactly what Fred Brooks's more than three-decades-old observations would lead us to expect. The attraction of OSS development is its ability to get lots of people to work on a project, but that is also a weakness. In the absence of firm design and documentation standards, and the ability to enforce those standards, the quality of the code is likely to suffer. If Linus Torvalds and the core developers were no longer participating in the Linux kernel project, we would expect that the Linux kernel could be reliably enhanced and modified only if an accurate, precise, and up-to-date description of its architecture and interfaces were always available. Without this, the changes are not likely to maintain the

conceptual integrity needed to prevent deterioration of the software (Parnas 1994b).

Notes

1. For the purpose of this chapter, we will consider the "architecture" to be (1) the division of the software into modules, (2) the interfaces between those modules, and (3) the *uses relation* between the externally accessible programs of those modules (Parnas 1979; Parnas, Clements, and Weiss, 1985).

2. In some cases, revised interfaces are dictated by the most powerful party, not negotiated.

3. In fact, many who acquire Linux pay a (relatively small) price for a "distribution" and some pay additional amounts for support such as documentation and advice about that version of Linux.

4. We use the word *module* to refer to a work assignment for a programmer or team (Parnas 1979).

5. The *interface* between two modules comprises all of the information about one module that would be needed to verify that the other was correct. Information about a module that is not included in the interface is considered to be *internal* implementation information.

6. The protocol itself is, of course, the subject of specification documents. However, even correct implementations of the protocol can differ internally. We were looking for specific documents of the Linux code components.

7. 1/3 of them are simple inline access functions.