성당과 시장(The Cathedral and the Bazaar)

에릭 레이먼드(Eric S. Raymond) $Date: 1998/05/13 17:29:31 $

번역: 정직한 honest@hitel.net, 1차 교정: 권순선 cessi@kldp.linux-kr.org, SGML 편집 및 2차 교정: 한지호 hajiho@penta.co.kr, 최종교정 : 정직한 honest@hitel.net 1998년 9월 10일

성공적인 오픈 소스 프로젝트인 fetchmail을 분석한다. 이 프로젝트는 리눅스의 역사에 의해 제시된 놀라운 소프트웨어 엔지니어링 이론을 신중하게 테스트하기 위해 실행된 것이다. 이 이론들을 두 개의 근본적으로 다른 개발 스타일의 용어들로 논할 것이다. 두가지 스타일이란 상업용 소프트웨어의 "성당" 모델과 리눅스 세계의 "시장" 모델이다. 이 모델들은 소프트웨어 디버깅 작업의 본질에 대한 서로 대립되는 가설들로부터 파생되었다는 것을 보일 것이다. 그리고 나서 리눅스의 경험으로부터 "충분히 많은 사람이 있다면, 찾을 수 없는 버그란 없다" 는 일관된 주장을 펴고, 이기적인 에이전트의 자가수정 시스템과의 생산적인 비유를 제시한 다음, 소프트웨어의 미래를 위해 이 통찰이 가지는 의미에 대한 탐구로 마무리짓는다.

목 차

제 1 절 성당과 시장 **1**

제 2 절 메일은 배달되어야만 한다 **2**

제 3 절 사용자가 있다는 것의 중요성 **4**

제 4 절 일찍, 그리고 자주 발표하라 **5**

제 5 절 장미가 장미다우려면 **7**

제 6 절 Popclient가 Fetchmail이 되다 **8**

제 7 절 Fetchmail 의 성장 **10**

제 8 절 Fetchmail에서 배울 점 **11**

제 9 절 시장 스타일의 개발에 필요한 선행조건들 **12**

제 10 절 오픈 소스 소프트웨어의 사회적 문맥 **13**

제 11 절 감사의 글 **15**

제 12 절 읽어볼 만한 글들 **15**

제 13 절 후기 : 넷스케이프가 시장 스타일을 받아들이다! **16**

제 14 절 버전과 변경 이력 **17**

제 1 절 성당과 시장

리눅스는 파괴적이다. 파트타임으로 해킹을 하면서 인터넷이라는 가느다란 선만으로 연결되어 있는 전세계 수천명의 개발자들에 의해 세계적인 수준의 운영체제가, 마치 마술처럼 만들어질 수 있었으리라고 누가 5년 전에 감히 상상이나 할 수 있었을까? 나는 분명 상상하지 못했다. 1993년 초, 리눅스가 내 레이다 화면에 잡혔을 때 나는 이미 유닉스와 오픈 소스 개발을 10년 동안 해오고 있었으며 1980년대 중반에 GNU 에 공헌한 첫 번째 사람들 중 한명이었다. 나는 네트워크 상에 꽤 많은 오픈 소스 소프트웨어를 발표했고, 지금도 널리 사용되고 있는 몇몇 프로그램을 개발중이거나 공동개발하고 있었다. (네트핵, Emacs VC 와 GUD 모드, xlife, 등등) 나는 프로그램이 어떻게 개발되어야 하는지 알고 있다고 생각했다.

리눅스는 내가 알고 있다고 생각한 많은 부분을 뒤집어 버렸다. 몇 년 동안이나 나는 작은 도구, 빠른 프로토타이핑, 그리고 진화적인 프로그래밍을 여러 해 동안 유닉스의 복음으로 설교해 오고 있었다. 하지만 나는 어떤 종류의 매우 중요한 복잡성이 있어서 거기에는 더 집중되고 선험적인 접근방법이 필요하다고 믿고 있었다. 가장 중요한 소프트웨어 (운영체제나 Emacs 같이 대단히 커다란 도구들)는 성당을 건축하듯이, 즉 찬란한 고독 속에서 일하는 몇 명의 도사 프로그래머나 작은 그룹의 뛰어난 프로그래머들에 의해 조심스럽게 만들어지고 때가 되기 전에 발표되는 베타버전도 없어야 한다고 생각했던 것이다.

리누스 토발즈의 개발 스타일은 - 일찍, 그리고 자주 발표하며 다른 사람들에게 위임할 수 있는 것은 모두 위임하고, 뒤범벅이 된 부분까지 공개하는 그런 스타일 - 나에게 놀라움으로 다가왔다. 고요하고 신성한 성당의 건축방식은 여기에서 찾아볼 수 없었다. 대신, 리눅스 공동체는 서로 다른 의견과 접근방법이 난무하는 매우 소란스러운 시장같았다. (리눅스 아카이브 사이트가 이것을 적절히 상징하고 있다. 이곳에는 *누구나* 파일을 올릴 수 있다) 이런 시장바닥에서 조리있고 안정적인 시스템이 나온다는 것은 거듭되는 기적에 의해서만 가능한 것처럼 보였다.

시장 스타일이 매우 효과적이라는 사실은 분명 충격이었다. 리눅스 공동체에 익숙해져 가면서 나는 개개의 프로젝트에 열심이었을 뿐만 아니라 왜 리눅스 세계가 공중분해 되지도 않고 성당건축가들이 상상하기도 힘든 속도로 계속해서 강해지는지 이해하려고 애썼다.

1996년 중반에야 이해가 되기 시작했다. 내 이론을 시험해 볼 수 있는 완벽한 기회가 오픈 소스 프로젝트의 형태로 찾아왔다. 여기에서 나는 의식적으로 시장 스타일을 시도해 볼 수 있었고, 큰 성공을 거두었다.

이 글의 나머지 부분에서는 그 프로젝트에 대해 이야기하고 효과적인 오픈 소스 개발에 대한 격언들을 제시할 것이다. 내가 이 모든 것을 리눅스 세계에서 처음 배운 것은 아니지만 리눅스 세계는 이 격언들이 특별한 의미를 가질 수 있게 해주었다. 만일 내가 옳다면, 독자들은 이 격언들로부터 리눅스 공동체가 훌륭한 소프트웨어를 만들어내는 원천이 될 수 있었던 이유를 이해할 수 있을 것이며, 독자들 자신도 더 생산적으로 되는 데 도움을 받을 수 있을 것이다.

제 2 절 메일은 배달되어야만 한다

1993년에 나는 펜실베니아 주, 서 체스터(West Chester) 시의 자그마한 무료 ISP인 체스터 카운티 인터링크 (Chester County InterLink : CCIL) 에서 기술적인 측면을 담당하고 있었다. (나는 CCIL 의 공동설립자였으며 우리만의 멀티유저 게시판 소프트웨어를 작성했다 - *locke.ccil.org* `<telnet://locke.ccil.org>`에 telnet 으로 접속하면 볼 수 있으며 지금은 19 회선으로 3000 여명의 사용자를 지원한다) 이 일 덕분에 나는 하루 24시간 내내 CCIL 의 56K 회선을 통해 네트워크에 접속해 있을 수 있었다 – 사실, 그렇게 해야만 하는 상황이었다.

그래서 나는 바로바로 배달되는 인터넷 이메일에 매우 익숙해져 있었는데 몇가지 복잡한 이유들로 인해 내 집의 컴퓨터 (snark.thyrsus.com) 과 CCIL 사이에 SLIP 연결을 하기가 꽤 힘들었다. 마침내 성공하고 나자, 주기적으로 locke 에 접속해 메일이 왔는지 체크해 보는 것이 매우 귀찮은 일이라는 것을 알게 되었다. 내가 원하는 것은 내 메일이 snark 로 배달되어 도착하는 즉시 내가 그것을 알 수 있고, 내 컴퓨터의 도구들을 이용해 메일을 다룰 수 있게 되는 것이었다.

sendmail을 이용해 단순히 포워드시키는 것은 소용이 없었다. 내 개인 컴퓨터가 항상 네트워크에 연결되어 있는 것도 아니고 고정적인 IP 어드레스를 가지고 있지도 않았다. SLIP 연결이 되면 내 메일을 가져와 내 컴퓨

터 안에서 배달해주는 프로그램이 필요했다. 그런 프로그램이 몇 개 있었고, 대부분은 프로토콜로 POP (Post Office Protocol)을 사용했다. 물론, locke 의 BSD/OS 운영체제에는 POP3 서버가 포함되어 있었다.

하지만 내게 필요한 것은 POP3 클라이언트였다. 그래서 네트워크를 뒤져 하나를 찾아냈다. 사실 서너개를 찾아내긴 했다. 잠시동안은 pop-perl을 사용했지만 기본적인 기능이 빠져 있었다. 가져온 메일에서 발신인의 주소를 제대로 처리하지 못해 답장을 보낼 수가 없었던 것이다.

문제는 이런 것이었다. locke 의 사용자 중에 'joe' 라는 사람이 나에게 메일을 보냈다고 해보자. snark 로 메일을 가져와서 그 메일에 답장을 하려고 하면 메일 프로그램은 snark 에는 있지도 않은 'joe' 에게 답장을 보내려고 시도한다. 그래서 손으로 '@ccil.org'를 답장 받는 사람의 주소 뒤에 붙여주어야 했는데, 이것은 곧 매우 피곤한 일이 되어버렸다.

이런 일은 분명히 컴퓨터가 해주어야 하는 일이었다. 하지만 이미 있는 POP 클라이언트들 중에서는 어느것도 이 일을 해주지 못했다. 여기에서 첫 번째 교훈을 얻을 수 있다.

1. 모든 좋은 소프트웨어는 개발자 개인의 가려운 곳을 긁는 것으로부터 시작된다. (Every good work of software starts by scratching a developer's personal itch)

명확해 보이는 교훈이긴 하지만 ("필요는 발명의 어머니" 라는 오래된 속담이 있지 않은가) 소프트웨어 개발자들은 너무나 자주, 단지 돈 때문에 그들이 필요로 하지도 않고 좋아하지도 않는 프로그램을 만들어 내는데 시간을 쓰고 있다. 하지만 리눅스 세계에서는 그렇지 않다 - 아마도 이것이 왜 리눅스 공동체에서 만들어진 소프트웨어들의 평균적이 품질이 그렇게나 좋은지를 설명해줄 것이다.

그래서 내가 이미 있는 POP3 클라이언트들과 경쟁하는 새로운 프로그램을 곧바로 코딩하기 시작했을까? 천만에. 나는 이미 가지고 있는 POP 유틸리티들을 조심스럽게 살피면서 스스로에게 물었다. "내가 원하는 것과 가장 가까운 프로그램이 어느 것일까?" 그 이유는

2. 좋은 프로그래머는 어떤 프로그램을 만들어야 할 지 안다. 위대한 프로그래머는 어떤 프로그램을 다시 만들어야 할 지 (그리고 재사용해야 할 지) 안다. (Good programmers know what to write. Great ones know what to rewrite(and reuse))

내가 위대한 프로그래머라는 말은 아니지만 흉내내려고는 했다. 위대한 프로그래머의 중요한 특징 중 하나는 건설적인 게으름이다. 그들은 들인 노력으로가 아니라 결과로 평가받는다는 것을 알고 있으며 완전한 무에서 시작하는 것보다는 부분적으로나마 좋은 해결책에서 시작하는 것이 거의 항상 더 쉽다는 것을 알고 있다.

리누스 토발즈 `<http://www.earthspace.net/~esr/faqs/linus>` 를 예로 들자면 그는 맨바닥에서 Linux를 만들어 내려고 하지 않았다. 대신 그는 386 기계를 위한 Unix 비슷한 소형 OS, Minix 의 코드와 아이디어를 재사용하는 것으로부터 시작했다. 결국 모든 Minix 코드는 사라지거나 새로 쓰여졌다 – 하지만 Minix 의 코드가 남아있을 동안 그 코드는 나중에 Linux 가 될 어린 아기의 발판 역할을 했다.

똑같은 생각으로 나는 이미 있는 POP 유틸리티 중 코딩이 잘 되어있는 것을 찾아 개발의 기초로 사용하려 했다.

Unix 세계의 소스를 공유하는 전통은 언제나 코드 재사용에 대해 호의적이었다. (GNU 프로젝트가 Unix 자체에 대한 심각한 의혹에도 불구하고 Unix 를 기본 OS 로 선택한 것도 바로 이런 이유에서였다) 리눅스 세계는 거의 기술적인 한계에 다다를 때까지 이 전통을 받아들였다. 일반적으로 찾아볼 수 있는 오픈된 소스가 수 테라바이트에 달하는 것이다. 그래서 리눅스 세계에서는 다른 어느 곳에서보다 누군가의 거의 완성된 소스를 찾아보는데 시간을 들이는 것이 좋은 결과를 가져다 줄 가능성이 높다.

나에게도 역시 그랬다. 예전에 찾아놓은 것에다가 두 번째 검색결과를 더하니 모두 아홉 개의 후보가 생겼다. fetchpop, PopTart, get-amil, gwpop, pimp, pop-perl, popc, popmail, 그리고 upop 이었다. 내가 제일 먼저 정착한 프로그램은 오승홍 씨의 fetchpop 이었다. 헤더 재작성 기능과 더불어 몇몇 개선사항을 추가했고, 저자가 릴리즈 1.9 에 그것을 수용했다.

몇 주 후에 나는 Carl Harris 가 만든 popclient 의 코드를 들여다 보다가 문제점을 발견했다. fetchpop 에는 훌륭한 독창적인 아이디어가 들어 있었지만 (daemon 모드 같은 것) POP3 만을 처리할 수 있었고, 아마추어 티가 나는 코딩이었다. (오승홍 씨는 똑똑하기는 하지만 경험이 부족한 프로그래머였으며 그 두 가지 특징 모두를 코딩에서 볼 수 있었다) Carl 의 코드는 전문가가 만든 탄탄하면서 더 나은 코드였으나 몇가지 중요하면서

도 구현하기 위해서는 약간의 잔머리가 필요한 fetchpop 의 기능들이 (내가 추가한 기능들을 포함해서) 빠져 있었다.

머물러 있을 것인가, 옮겨갈 것인가? 옮겨간다면 더 나은 개발기반을 위해 이미 해놓은 코딩을 포기해야만 했다.

옮겨가는데 실질적인 동기가 되었던 것은 다중 프로토콜 지원 여부였다. POP3 가 우체국 서버 프로토콜 중에서 가장 널리 쓰이는 것이긴 했지만 유일한 프로토콜은 아니었다. fetchpop 을 비롯하여 다른 경쟁자들은 POP2, RPOP, 또는 APOP 를 지원하지 않았고, 나는 당시에 재미삼아서 *IMAP* <http://www.imap.org> (Internet Message Access Protocol, 가장 최근에 고안되었으며 가장 강력한 우체국 프로토콜) 을 지원해 볼까 하는 생각을 가지고 있었다.

하지만 옮겨가는 것이 좋은 생각이라는 좀 더 이론적인 이유도 가지고 있었다. 리눅스를 알기 오래전에 배운 교훈이었다.

3. "가지고 있는 것을 버릴 계획을 세우라 ; 언젠가는 버리게 될 것이다 (Plan to throw one away; youu will anyhow)" (Fred Brooks, "The Mythical Man-Month", Chapter 11)

다른 말로 하자면, 첫 번째 해결책을 구현할 때까지도 진짜 문제가 무엇인지 이해하지 못하는 경우가 종종 있다는 것이다. 두 번째가 되어서야 어떻게 하는 것이 옳은 것인지 충분히 알게 될 수 있다. 따라서 만일 올바른 방법을 찾고 싶다면 *최소한* 한 번은 처음부터 다시 시작할 준비를 해 두어야 한다. 그래, fetchpop을 고친 것은 내 첫 번째 시도였어, 하고 스스로에게 말하고 나서 나는 popclient 로 옮겨갔다.

1996년 6월 25일에 Carl Harris 에게 내 첫 번째 popclient 패치를 보낸 후, 나는 그가 popclient 에 대한 흥미를 이미 잃었다는 것을 알게 되었다. 코딩이 좀 지저분했고, 자잘한 버그들이 널려있었다. 내가 수정해야 할 것이 많았고, Carl 과 나는 곧 내가 프로그램을 넘겨받는 것이 합리적이라는 데에 동의하게 되었다. 내가 알아차리지 못하는 새에 프로젝트가 차츰 궤도에 오르기 시작했다. 나는 이미 존재하고 있는 POP 클라이언트의 마이너 패치를 생각하는 것이 아니었다. 클라이언트 하나를 통채로 관리하고 있었으며 내 머리에서는 커다란 변화가 될 아이디어들이 솟아나고 있었다.

코드 공유를 장려하는 소프트웨어 문화에서는 이런 방식으로 프로젝트가 진화하기 마련이다. 이렇게 말할 수 있다.

4. 적절한 태도를 가지고 있으면 흥미로운 문제가 당신을 찾아갈 것이다. (If you have the right attitude, interesting problems will find you)

하지만 Carl Harris 의 태도가 훨씬 더 중요했다. 그의 이것을 이해하고 있었다.

5. 프로그램에 흥미를 잃었다면 프로그램에 대한 당신의 마지막 의무는 능력있는 후임자에게 프로그램을 넘겨주는 것이다. (When you lose interest in a program, your last duty to it is to hand it off to a competent successor)

토론할 필요도 없이 Carl 과 나는 우리가 가장 좋은 해결책을 찾고 있다는 것을 알고 있었다. 우리에게 남아있는 한가지 문제는 내가 적임자라는 것을 입증할 수 있느냐 하는 것이었다. 내가 그것을 증명하자 그는 기꺼이, 그리고 신속하게 행동했다. 내가 그렇게 행동할 차례가 되었을 때 나도 그만큼 잘 할 수 있기를 바란다.

제 3 절 사용자가 있다는 것의 중요성

그래서 내가 popclient를 넘겨 받았다. 내가 popclient 의 사용자들을 넘겨받았다는 것도 그에 못지않게 중요하다. 사용자들이 있다는 것은 매우 좋은 일이다. 당신이 누군가의 필요를 충족시켜주고 있으며 일을 잘 해나가고 있다는 것을 보여주기 때문만은 아니다. 적절하게 유도해 준다면 사용자들은 공동개발자가 될 수도 있다.

유닉스의 전통이 가지고 있는 또하나의 강점, 즉 많은 수의 사용자들이 동시에 해커이기도 하다는 것을 리눅스는 좋은 의미로서의 극단까지 밀어붙였다. 소스코드가 공개되어 있기 때문에 그들은 *효과적인* 해커가 될 수 있다. 이것은 디버깅 시간을 줄이는 데 엄청난 도움이 되었다. 조금만 격려해주면 사용자들은 문제를 분석하고 해결책을 제시하며, 도움 없이 혼자 일할 때보다 훨씬 빨리 코드를 개선시키도록 해준다.

6. 사용자들을 공동개발자로 생각하면 코드가 다른 어떤 방법보다도 빠른 속도로 개선되며 효율적으로 디버깅할 수 있다. (Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging)

이 효과의 위력은 과소평가되기 쉽다. 사실, 오픈 소스 세계의 우리들조차 시스템의 복잡도에 대항하여 많은 수의 사용자가 얼마나 힘이 되는지를 리누스 토발즈가 보여주기 전까지는 과소평가하고 있었다.

실제로 나는 리누스의 가장 영리하고 가장 중요한 해킹은 리눅스 커널을 만들었다는 점이 아니라 리눅스 개발모델을 만들었다는 점이라고 생각한다. 리누스에게 이 의견을 말해 주었더니 그는 씨익 웃고서 조용히 여러번 하던 말을 되풀이했다. "난 기본적으로 매우 게으른 사람이라서 실제로는 다른 사람들이 해놓은 일을 가지고 공로라고 인정받곤 해요." 여우처럼 게으르군. 로버트 하인라인이라면 '실패하기에는 너무 게으르다' 고 말했을 것이다.

되돌아 보면, 리눅스의 성공과 방법론은 GNU Emacs Lisp 라이브러리와 Lisp 코드 아카이브에서 그 선례를 찾아볼 수 있다. Emacs C 코어와 다른 대부분의 FSF 도구들의 성당건축 스타일과는 대조적으로 Lisp 코드 풀의 진화는 유동적이었고, 사용자가 주도한 것이었다. 아이디어와 프로토타입 모드들은 안정적인 최종형태를 갖추기까지 종종 서너번씩 다시 쓰여졌다. 느슨하게 묶인 공동작업이 인터넷으로 인해 가능해졌고, 리눅스에서처럼 매우 자주 일어나는 일이 되었다.

사실 fetchmail 이전에 내 자신의 가장 성공적인 해킹은 아마 Emacs VC 모드였을 것이다. 세 명의 사람들과 email을 통해 리눅스와 비슷한 협동작업을 했고, 지금까지 그 셋 중의 한 명 (리차드 스톨만:Richard Stallman. Emacs 의 저자이면서 *FSF* `<http://www.fsf.org>` 의 설립자) 만을 만나보았다. VC 모드는 SCCS, RCV 와 CVS 를 위한 Emacs 내의 프론트엔드였고, "원터치" 버전컨트롤 기능을 제공했다. 이것은 누군가 만들어 놓은 작고 조악한 sccs.el 모드로부터 진화한 것이었다. VC 의 개발은 Emacs 와는 다르게 Emacs Lisp 코드가 발표/테스트/개선의 주기를 매우 빨리 반복할 수 있었기 때문에 성공했다.

코드를 법적으로 GPL 에 묶어두려는 FSF 의 정책은 한가지 예기치 못한 부작용을 가져왔는데, 그것은 FSF 가 시장모드를 사용하는 것이 절차적으로 까다로워졌다는 것이다. 그들은 GPL 코드를 저작권법 하에서의 도전으로부터 면역시키기 위해 20줄 이상의 개인적인 공헌에 대해서는 저작권을 주어야 한다고 믿기 때문이다. BSD 와 MIT 의 X 콘소시엄 라이센스를 사용하여 저작권을 얻는 사람들에게는 이런 문제가 없다. 그들은 누군가가 도전할 동기를 가질만한 권리를 가지려 하지 않는다.

제 4 절 일찍, 그리고 자주 발표하라

일찍, 그리고 자주 발표하는 것은 리눅스 개발 모델의 중요한 부분이다. 대부분의 개발자들은 (필자를 포함하여) 아주 사소한 프로젝트가 아니라면 이런 정책은 나쁜 것이라고 생각했다. 초기버전들은 예외없이 버그가 많고, 개발자라면 사용자들의 인내심을 시험하고 싶지는 않기 때문이다.

이런 믿음이 성당건축 스타일의 개발을 더 선호하게 만들었다. 만일 가장 중요한 목표가 사용자들로 하여간 가능한 한 적은 버그를 발견하게 만드는 것이라면 6 개월에 한 번씩 (혹은 그보다 더 늦게) 발표하면서 그동안 죽어라고 일하는 편이 나을 것이다. Emacs C 코어는 이런 식으로 개발되었다. Lisp 라이브러리는 그렇지 않았다. Emacs 의 발표주기와 관계없이 언제든 새로운 개발 코드 버전을 찾을 수 있으며, FSF 의 통제권 밖에 있는 Lisp 라이브러리들이 있었기 때문이다.

이들 중 가장 중요한 아카이브는 오늘날 대형 리눅스 아카이브들의 정신과 많은 기능들을 이미 가지고 있었던 오하이오 주의 elisp 아카이브였다. 하지만 우리가 하고 있는 일에 대해, FSF 의 성당건축 개발모델의 문제점들에 대해 그 아카이브의 존재가 무엇을 제시하는지에 대해 우리들 중 소수만이 진지하게 생각하고 있었다. 나는 1992년에 오하이오 코드를 공식적인 Emacs Lisp 라이브러리에 정식으로 병합 시키려는 시도를 했으나 정치적인 문제에 부딪쳤고, 큰 실패를 겪었다.

1년 후에, 리눅스가 널리 알려지기 시작했고, 무언가 다르면서도 훨씬 바람직한 일이 일어나고 있다는 것이 확실해 보였다. 리누스의 열린 개발정책은 성당건축과 완전히 반대되는 것이었다. 선사이트와 tsx-11 아카이브가 싹트고 있었고, 다중배포방식이 퍼지기 시작했다. 그리고 이 모든 것이 이전의 어느 소프트웨어보다 자주 릴리즈되는 코어시스템에 의해 주도되고 있었다.

리누스는 가장 효과적인 방식으로 사용자들을 공동개발자라고 여겼던 것이다.

7. 일찍 발표하고 자주 발표하라. 그리고 사용자들의 소리에 귀를 기울이라. (Release early. Release often. And listen to your customers)

리누스의 혁신은 그가 이렇게 했다는 점 보다는 (그 비슷한 것이 오랫동안 유닉스 세계의 전통이었다) 그가 개발하고 있던 리눅스 커널의 복잡성에 비견될만한 수준으로까지 끌어올렸다는 데 있다. 초기에 (1991년 경에) 그는 *하루에* 한 번 이상 새로운 커널을 발표하기까지 했다. 리누스가 공동개발자들이라는 자신의 기반을 잘 만들었고, 인터넷이라는 지렛대를 이용하여 누구보다도 열심히 협동작업에 몰두했기 때문에 이런 방식은 성공했다.

하지만 *어떤 과정을 거쳐* 성공할 수 있었을까? 내가 재현할 수 있는 것일까, 아니면 리누스 토발즈만의 천재성이 필요한 것일까?

그렇게 생각되지는 않았다. 리누스가 매우 뛰어난 해커라는 점은 인정한다. (우리중에 상업용 제품 못지 않은 운영체제의 커널을 만들어낼 수 있는 사람이 몇이나 될까?) 하지만 리눅스는 놀랄만한 개념적 전진을 이루어내지는 않았다. 리누스는 리차드 스톨먼이나 제임스 고슬링 (NeWS 와 자바를 만든) 과 같은 혁신적인 설계를 이루어내는 천재는 (적어도 지금까지는) 아니었다. 대신 리누스는 공학의 천재인 것으로 보인다. 버그와 개발의 막다른 골목을 피하는 육감, 그리고 A 점에서 B 점까지 가는데 최소노력 경로를 찾아내는 요령을 갖추고 있었다. 실제로 리눅스의 전반적인 설계는 이런 특성을 바탕으로 하고 있으며 리누스의 본질적으로 보수적이고 단순한 설계 방식을 반영하고 있다.

따라서 빠른 릴리즈와 인터넷을 매체로 사용하는 것이 우연히 이루어진 것이 아니라 리누스의 공학적 천재성에 기인한 최소노력 경로에 대한 통찰력의 통합적인 부분이었다면 그가 최대화하고 있는 것은 무엇이었을까? 기계에서 무엇을 뽑아내었던 것일까?

해답은 질문 안에 있다. 리누스는 그의 해커/사용자들에게 지속적인 자극과 보답을 제공했다 – 리눅스 개발에 참여함으로써 자기만족을 얻으리라는 전망에 자극받았고, 그들이 하는 일이 계속해서 (어떤 때는 *날마다*) 향상되고 있다는 것이 보답이 되었다.

리누스는 만일 처리하기 곤란한 심각한 버그가 발견되면 사용자들이 떨어져 나갈 위험과 코드가 불안정해질 가능성을 무릅쓰고 디버깅과 개발에 투입되는 공수(the number of person-hours)를 최대화 하는 것에 목표를 두었다. 리누스는 다음과 같은 신념을 가지고 있는 것처럼 행동했다.

8. 충분히 많은 베타테스터와 공동개발자가 있으면 거의 모든 문제들은 빨리 파악될 것이고 쉽게 고치는 사람이 있게 마련이다. (Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone)

덜 형식적으로 말하자면, "보고 있는 눈이 충분히 많으면 찾지 못할 버그는 없다." 나는 이것을 "리누스의 법칙" 이라고 부른다.

내 원래의 공식적인 서술은 모든 문제는 "누군가에게는 간단할 것이다" 였다. 리누스는 문제를 이해하고 고치는 사람이 그 문제를 처음 파악한 사람과 항상 같은 것이 아니라 오히려 다른 경우가 더 많다고 이의를 제기했다. 리누스의 얘기로는, "누군가 문제를 발견합니다. 그리고 *또다른* 누군가가 그 문제를 이해하지요. 문제를 발견해 내는 것이 더 중요한 일이라고 분명히 말할 수 있습니다." 하지만 가장 중요한 점은 사람이 충분히 많을 경우 이 두 가지가 모두 매우 빨리 일어나는 경향이 있다는 것이다.

내 생각에는 여기에 성당 건축과 시장 스타일의 핵심적인 차이점이 있다. 프로그래밍의 성당 건축가 관점에서 보자면 버그와 개발 문제는 어렵고, 까다로우며 심오한 현상이다. 문제를 해결하려면 헌신적인 소수의 사람이 몇 달이고 정밀한 검사를 수행해야 모두 끝났다는 확신을 가질 수 있다. 따라서 발표 사이의 기간이 길어지고, 오랫동안 기다린 릴리즈가 완벽하지 않을 때는 필연적으로 실망이 따른다.

반면, 시장의 관점에서는 버그가 보통 쉽게 해결될 수 있는 것이라고 본다 – 최소한 새로운 릴리즈가 나올때마다 그것과 씨름하는 수천의 열정적인 공동개발자들에게 알려진다면 금방 쉽게 해결할 수 있는 문제로 바뀐다. 따라서 더 많이 교정을 받고 싶다면 자주 발표해야 하며 덤으로 서투른 부분이 드러나더라도 잃을 것이 적다는 이점이 있다.

바로 이것이다. 이것으로 충분하다. "리누스의 법칙" 이 틀렸다면 리눅스 커널과 같이 복잡한 시스템은 어떤 것

이라도 수많은 손들에 의해 해킹되면서 일찍이 볼 수 없었던 나쁜 상호작용과 발견되지 못한 "심오한" 버그들에 의해 어느 시점에선가 붕괴되고 말았을 것이다. 반면에, 만일 그 법칙이 옳다면 그 법칙만으로도 리눅스의 상대적으로 적은 버그를 설명할 수 있다.

그리고 이 법칙이 옳다는 것에 대해서 너무 놀라지 말아야할 것이다. 수년 전, 사회학자들은 비슷하게 전문적인 (혹은 비슷하게 무지한) 관찰자들로 이루어진 대중의 평균적인 의견이 그 관찰자 중 무작위로 뽑은 한 명의 의견보다 더 신뢰할 만하다는 점을 발견했다. 사회학자들은 이것을 "델파이 효과" 라고 부른다. 리누스가 보여준 것은 이 효과가 운영체제를 디버깅하는 데에도 적용될 수 있다는 점이다. 델파이 효과는 OS 커널만큼 복잡한 개발까지도 다룰 수 있는 것이다.

고맙게도 제프 덧키(Jeff Dutky) <dutky@wam.umd.edu> 는 리누스의 법칙을 "디버깅은 병렬처리가 가능하다" 는 말로 표현할 수 있음을 지적해 주었다. 제프는 디버거들이 디버깅을 하려면 의사소통을 조정해주는 개발자가 필요하지만 디버거들 사이에는 그다지 조정이 필요하지 않다고 진술한다. 따라서 개발자를 추가하는데서 생기는 기하급수적인 복잡성과 관리의 어려움이 디버깅에는 짐이 되지 않는다.

실제로 리눅스 세계에서는 디버거의 작업이 중복됨으로써 생기는 이론적인 효율 저하가 거의 문제되었던 적이 없는 것으로 보인다. "빨리, 그리고 자주 발표하는 정책" 의 효과 중 하나는 피드백되어 오는 수정사항을 빨리 전파함으로써 중복이 최소화된다는 것이다. 브룩스(Brooks)는 제프의 진술과 관련하여 즉석에서 다음과 같은 말을 했다. "널리 사용되는 프로그램의 유지보수에 들어가는 비용은 보통 개발시 드는 비용의 40 퍼센트나 그 이상입니다. 놀랍게도 이 비용은 사용자의 수에 큰 영향을 받습니다. *더 많은 사용자들이 더 많은 버그를 찾아냅니다.*"

사용자들이 많아지면 프로그램을 시험해보는 방법이 더 늘어나기 때문에 버그를 더 많이 잡아낼 수 있다. 이 효과는 사용자들이 공동개발자들일 때 더욱 커진다. 각 사람들이 버그를 찾아낼 때 조금씩 다른 개념의 집합과 분석 도구들을 사용하여 문제의 다른 각도에서 접근하기 때문이다. "델파이 효과" 는 바로 이런 편차에서 비롯되는 것으로 보인다. 또한 디버깅이라는 특정한 환경에서 이 편차는 노력의 중복을 줄여주는 경향이 있다.

따라서 더 많은 베타테스터를 가지는 것은 *개발자*의 관점에서 현재의 "가장 심오한" 버그의 복잡성을 줄여주지는 않을 테지만, 누군가의 도구가 문제에 딱 들어맞아 그 버그가 *그 사람에게*는 쉽게 잡을 수 있는 것이 될 가능성을 높여준다.

리누스도 물론 할 일이 있었다. 심각한 버그가 있을 경우에 대비해 리눅스 커널 버전은 잠재 사용자들이 최종적으로 "안정된" 버전을 사용할 수도 있고 새로운 기능을 사용하기 위해 최신의 버그가 있을 수 있는 버전을 사용할 수도 있게 번호가 붙여졌다. 이 전술은 아직까지 대부분의 리눅스 해커들이 따라하지는 않고 있지만 아마도 따라하게 될 것이다. 두 가지 선택이 가능하다는 사실이 양쪽 모두를 더 매력적으로 보이게 한다.

제 5 절 장미가 장미다우려면

리누스의 행동을 연구하고 그것이 왜 성공적이었는지에 대한 이론을 만든 후, 나는 이 이론을 내 새로운 프로젝트 (물론 훨씬 덜 복잡하고 덜 야심적인 프로젝트) 에 적용해 보기로 했다.

그러나 내가 가장 먼저 한 일은 popclient 를 더 재조직화하고 단순화한 것이었다. 칼 해리스 (Carl Harris) 의 구현방식은 매우 건강한 것이었지만 많은 C 프로그래머들에게서 볼 수 있었던 것처럼 일종의 불필요한 복잡성을 보여주고 있었다. 그는 코드를 중심적인 것으로, 자료구조는 코드를 받쳐주는 것으로 취급했다. 그 결과 코드는 아름답지만 자료구조는 임시변통(ad-hoc)으로 설계되었고, 보기에 좋지 않았다. (최소한 옛 LISP 해커의 높은 기준에서 보자면 말이다)

그리고 코드와 자료구조를 개선하는 것 말고도 나는 또다른 목적을 가지고 있었다. 그것은 popclient를 내가 완전히 이해하는 무엇인가로 진화시키는 것이었다. 이해하지 못하는 프로그램의 버그를 수정하는 책임을 맡는 것은 괴로운 일이다.

처음 한달 정도가 지날 동안 나는 그저 칼의 기본적인 설계가 어떤 의미를 가지고 있는지 따라다니기만 했다. 내가 처음으로 중요한 수정을 가한 것은 IMAP 지원이었다. 프로토콜 머신을 일반적인 드라이버와 세가지 메소드 테이블 (POP2, POP3, IMAP을 지원하는) 로 재조직했다. 이것과 그 이전의 변경들은 프로그래머들이 기억해 둘만한 일반적인 원리를 보여준다. 특히 C 와 같이 즉흥적으로 프로그램하기 힘든 언어에서는.

9. 자료구조를 훌륭하게 만들고 코드를 멍청하게 만드는 것이 그 반대의 경우보다 훨씬 잘 작동한다. (Smart data structures and dumb code works a lot better than the other way around)

브룩스의 책 9 장(Chapter 9) 에 이렇게 쓰여있다. "내게 [코드]를 보여주고 [자료구조]를 숨긴다면 나는 계속 어리둥절할 것이다. 자료구조를 보여준다면 코드는 볼 필요도 없이 뻔한 것이다." 사실 브룩스는 "흐름도" 와 "테이블" 이라고 이야기했다. 하지만 30년간 변해온 용어들과 문화를 고려한다면 거의 똑같은 말이라고 할 수 있다.

이 시점에서 (1996년 9월 초, 일을 시작하고 6 주가 지난 후) 나는 이름을 바꿀 때가 되었다고 생각하기 시작했다. 이 프로그램은 더 이상 POP 클라이언트만이 아니었다. 하지만 설계상에 정말 새로운 것이 들어가 있지 않았기 때문에 머뭇거리고 있었다. 내가 만든 popclient 는 아직 스스로의 정체성을 확립하지 못하고 있었다.

fetchmail 이 어떻게 SMTP 포트로 가져온 메일을 포워드 시켜야 하는지 알고 난 후에는 상황이 급변했다. 그에 대해서는 잠시 후에 이야기할 것이다. 하지만 그보다 먼저, 앞서 나는 리누스 토발즈가 옳은 방법으로 일을 해냈다는 내 이론을 시험하기 위해 이 프로젝트를 수행하기로 했다고 말했다. 어떻게 시험을 했을까? 다음과 같은 방법을 사용했다.

1. 일찍, 자주 발표했다. (발표간격이 10일을 넘는 경우는 거의 없었으며 개발에 몰두해 있을 때는 하루에 한번씩 발표했다)
2. fetchmail 에 대한 일로 나에게 연락해 오는 사람은 누구든지 베타테스터 목록에 올렸다.
3. 새로 발표할 때마다 베타테스터들에게 떠들썩하게 발표를 알리며 사람들이 참여하도록 격려했다.
4. 그리고 그들의 이야기를 들었다. 설계 결정에 대해 투표를 하기도 했고 패치나 피드백을 보내올 때마다 베타테스터들을 구슬렀다.

이 단순한 방법들은 즉각 효력을 나타냈다. 프로젝트를 시작할 때부터 개발자들이라면 학수고대할 만한 버그 리포트를, 때로는 훌륭하게 수정된 코드를 받을 수 있었다. 사려깊은 비판과 팬 메일, 기능제안들을 받았다. 여기서 다음과 같은 결론을 이끌어낼 수 있다.

10. 베타테스터들을 가장 중요한 자원으로 여긴다면 그들은 정말 가장 중요한 자원이 되어준다. (If you treat your beta-testers as if the're your most valuable resource, they will respond by becoming your most valuable resource)

fetchmail 의 성공을 재는 재미있는 척도 중 하나는 프로젝트 베타테스터 메일링리스트인 fetchmail-friends 의 크기이다. 이 글을 쓰고있을 때 목록에는 249 명이 있었고 1주일에 2-3 명이 추가되었다.

1997 년 5월말 경에 글을 수정하면서 보니까 목록은 300명 가까이 되었고, 멤버들이 조금씩 줄기 시작했는데 그 이유가 흥미로왔다. 몇몇 사람들이 구독을 중단하면서 fetchmail 이 잘 작동하기 때문에 더 이상 메일링리스트를 보고 있을 이유가 없다고 말했다. 아마 이것이 성숙한 시장 스타일의 프로젝트가 가지는 정상적인 라이프사이클 중 하나일 것이다.

제 6 절 Popclient가 Fetchmail이 되다

fetchmail 프로젝트에서 큰 전환이 일어났던 것은 해리 호흐하이저(Harry Hochheiser) 가 클라이언트 머신의 SMTP 포트로 메일을 포워딩하는 대략적인 코드를 보내준 때였다. 보자마자 이 기능을 안정적으로 구현한다면 다른 모든 배달 방법은 구식이 되리라는 것을 깨달았다.

여러 주 동안 나는 fetchmail을 조금씩 뜯어고치고 있었는데, 인터페이스 설계가 작동하긴 하지만 지저분하다고 느끼고 있었다. 우아하지도 않고 몇 안되는 옵션들이 너무 여기저기 흩어져 있었다. 가져온 메일을 메일박스 파일에 부어놓을 것인지, 표준출력으로 내보낼 것인지 결정하는 옵션이 특히 골치거리였지만 왜 그런지 확실히 깨닫지는 못했다.

SMTP 포워딩을 생각하자 그동안 popclient 가 너무 많은 것을 해내려고 했다는 것을 알게 되었다. poopclient 는 MTA (Mail Transport Agent) 와 MDA (Mail Delivery Agent)의 기능을 모두 가지도록 설계되었다. SMTP

포워딩만 할 수 있다면 MDA 기능을 없애 순수한 MTA 가 될 수 있었다. sendmail 과 마찬가지로 최종적인 메일 배달은 다른 프로그램에게 맡기면 되는 것이다.

TCP/IP를 지원하는 플랫폼이라면 거의 어디에나 25번 포트가 기다리고 있는데 무엇 때문에 복잡한 MDA 기능을 설정하거나 메일박스를 잠그고 덧붙이는 (lock-and-append) 문제를 가지고 고생을 하는가? 더구나 포워딩을 사용하면 가져온 메일이 평범한 SMTP 메일처럼 보일 것이고, 우리가 원하는 것이 바로 그것이었는데 말이다.

몇가지 배울 점이 있었다. 먼저, SMTP 포워딩에 대한 아이디어는 내가 리누스의 방법을 모방하려고 의식적으로 노력한 것에 대한 가장 큰 보답이었다. 사용자 한 명이 내게 끝내주는 아이디어를 주었으며 내가 해야했던 일은 그 의미를 이해하는 것 뿐이었다.

11. 좋은 아이디어를 생각해내는 것 다음으로 중요한 일은 사용자들이 알려준 좋은 아이디어를 깨닫는 것이다. 때로는 이편이 더 나을 수도 있다. (The next best thing to having good ideas is recognizing good ideas from your users. Sometimes the latter is better)

흥미롭게도 만일 당신이 얼마나 다른사람에게 빚을 많이 지고 있는지를 자기 비하라고 느껴질 정도로까지 솔직하게 털어놓는다면 대개의 사람들은 당신이 혼자서 거의 모든 일을 해내고서 천재성에 대해서 겸손해 하는 것처럼 대한다는 것을 곧바로 알게 될 것이다. 리누스의 경우를 보라! (1997년 8월, Perl 컨퍼런스에서 이 글을 발표할 때 래리 월이 첫 번째 줄에 앉아 있었다. 바로 윗 줄에 도달했을 때 그는 부흥사라도 된 것처럼 외쳤다. "형제여, 이야기 하시오, 이야기를!" 청중들 모두가 이것이 Perl을 만든 래리에게도 적용된다는 것을 알았기 때문에 웃음을 터뜨렸다.)

똑같은 정신으로 프로젝트를 몇 주 진행해 나가자 나는 사용자들 뿐 아니라 이야기를 전해들은 다른 사람들로부터 비슷한 칭송을 받기 시작했다. 나는 그런 email 중 몇몇을 따로 보관해 두었다. 나중에 내 삶이 가치있는 것이었는지 의심스러워질 때 그 메일들을 다시 꺼내볼 생각이다. :-)

모든 종류의 설계에 대해서 적용될 수 있는 두가지 더 기본적이며 비정치적인 교훈이 있다.

12. 종종 가장 충격적이고 혁신적인 해결책은 당신 자신이 문제에 대해서 가지고 있는 개념이 잘못되어 있다는 것을 깨닫는 것에서 나온다. (Often, the most striking and innovative solutions come from realizing that your concept of the problem was wrong)

나는 popclient를 MTA/MDA 기능을 다 갖추고 복잡한 지역배달모드들까지 (local delivery modes) 갖춘 것으로 개발해 나가면서 틀린 문제를 풀려고 노력하고 있었다. fetchmail 의 설계는 가장 기초적인 것부터 재고하여 SMTP 포트로 메일을 배달하는 인터넷 메일 경로의 한 부분인 순수 MTA 가 되어야 했다.

개발 도중에 벽에 부딪친다면 - 다음번 패치 후에 무엇을 해야 할 지 모르겠다면 - 그때는 정답을 가지고 있는지 생각할 것이 아니라 질문이 올바른 것인지 의문을 가져보아야 하는 경우가 종종 있다. 아마도 문제의 틀을 다시 잡아야 할 것이다.

그래서, 나도 내 문제의 틀을 다시 잡았다. 분명히 제대로 일을 진행하려면 (1) SMTP 포워딩 지원 기능을 일반 드라이버에 포함시키고, (2) SMTP 포워딩을 기본모드로 만들고 (3) 최종적으로는 다른 배달모드들, 특히 '파일로 배달하기' 와 '표준출력으로 배달하기'를 제거해야 했다.

나는 단계 (3)에서 조금 머뭇거렸는데, 이유는 오랫동안 popclient 를 써오면서 다른 배달모드에 의존하고 있을 사용자들의 심기를 불편하게 만들고 싶지 않았기 때문이다. 이론적으로는 그들 모두 즉시 .forward 파일이나 sendmail 외의 비슷한 프로그램으로 전환하여 동일한 결과를 얻을 수 있었다. 실제로는 전환 자체가 큰 일이 될 것이었다.

하지만 단계 (3)을 실행하고 나자 이점이 매우 큰 것으로 나타났다. 드라이버 코드 중 가장 힘든 부분이 사라졌다. 설정이 엄청나게 간단해졌다 - 시스템의 MDA 와 사용자의 메일박스를 일일이 찾아다니며 굽실거릴 필요도 없어졌고, OS 가 파일 잠금을 지원하는지 걱정할 필요도 없어졌다.

게다가 메일을 잃어버릴 한가지 가능성도 사라졌다. '파일로 배달하기'를 선택했을 때 디스크가 꽉 차 있으면 메일이 사라져 버렸던 것이다. SMTP 포워딩에서는 SMTP 리스너가 메시지 배달이 가능하거나 나중에 배달할 수 있도록 스풀해 놓기 전에는 OK를 돌려주지 않을 것이기 때문에 이런 일이 일어날 수가 없다.

성능도 향상되었다(한두번 실행시켜서는 느끼지 못하겠지만). 또 변경에 따르는 그다지 중요하지 않은 이익이

라면 매뉴얼 페이지가 훨씬 간단해 졌다는 것이다. 나중에 나는 사용자가 지정한 지역 MDA를 통해 배달하는 기능을 다시 넣어야 했다. 동적인 SLIP를 포함하여 몇몇 애매한 상황을 다루어야 했기 때문이다. 하지만 처음보다 훨씬 간단한 방법을 찾아낼 수 있었다.

교훈이라면? 낡아서 사용할 수 없는 기능이라면 효율을 떨어뜨리지 않고 제거할 수 있을 때는 망설이지 말고 제거해 버리라. 앙뜨완 드 생떽쥐뻬리는 (아동서적 작가였으며 남는 시간에는 비행기 조종과 설계를 했던) 이렇게 말했다.

13. "(설계에 있어서) 완벽함이란 더 이상 추가할 것이 없을 때 이루어지는 것이 아니라 더 이상 버릴 것이 없을 때 이루어진다. (Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away)"

코드가 더 나아지고 간단해지고 있을 때가 바로 일이 제대로 되어가고 있다는 것을 *알게 되는* 때다. 그리고 그 과정에서 fetchmail 의 설계는 그 조상격인 popclient 와 다른, 자신만의 정체성을 획득했다. 이름을 바꿀 때가 된 것이다. 새로운 설계는 예전의 popclient 보다는 sendmail 과 비슷해 보였다. 둘다 MTA 였으나 sendmail 은 푸시(push) 후에 메일을 배달했고 새로운 popclient 는 풀(pull) 후에 메일을 배달했다. 해서 두 달 후에 나는 popclient 의 이름을 fetchmail 로 변경했다.

제 7 절 Fetchmail 의 성장

이제는 깔끔하고 혁신적인 설계, 매일 사용하므로 잘 작동하는 것을 알고 있는 코드, 발전하고 있는 베타테스터의 목록을 가지고 있었다. 더이상 내가 하고 있는 일이 몇몇의 사람에게 유용할 수도 있는 사소하고 개인적인 해킹은 아니라는 생각이 서서히 들기 시작했다. 내가 가지고 있는 것은 유닉스 박스와 SLIP/PPP 메일 연결을 가지고 있는 모든 해커들이 정말로 필요로 하는 프로그램이었다.

SMTP 포워딩 기능으로 fetchmail 은 경쟁에서 멀찍이 앞서나와 "카테고리 킬러," 그러니까 해당분야의 다른 프로그램들은 아예 잊혀져 버릴 만한 경쟁력을 갖추고 자신의 지위를 확고하게 하는 고전적인 프로그램이 될 수 있는 능력을 갖추었다.

이런 결과를 계획하거나 목표로 가질 수는 없으리라고 생각한다. 아주 강력한 설계상의 아이디어로 그런 결과가 불가피하고, 자연스러우며 운명적인 것으로 보이게 함으로써 그런 결과에 도달해야 한다. 그런 아이디어를 구체화해 볼 수 있는 유일한 방법은 수많은 아이디어를 가지는 것이다. 아니면 다른 사람들의 좋은 아이디어를, 원래 생각되었던 것보다 더 멀리 이끌고 가서 구체화 시켜보는 방법이다.

앤드류 타넨바움 (Andrew Tanenbaum) 은 교습 도구로 사용하기 위해 386용으로 간단한 네이티브 유닉스를 만들려는 원래의 아이디어를 가지고 있었다. 리누스 토발즈는 이 미닉스의 개념을 앤드류가 생각했던 것보다 더 멀리 밀고 나갔다. 그래서 리눅스는 굉장한 것이 되었다. 똑같은 방식으로 (더 작은 스케일이었지만) 나는 칼 해리스와 해리 호흐하이저의 아이디어들을 가져와 강하게 밀어붙였다. 리누스나 나나 사람들이 천재들이 그러하리라고 생각하는 낭만적인 의미에서 '독창적' 인 것은 아니었다. 하지만 통념과는 반대로 대부분의 과학과 공학과 소프트웨어 개발은 독창적인 천재, 해커의 전설에 의해서 이루어지지는 않는다.

결과물은 똑같이 매우 사람을 흥분시키는 것들이다 – 사실, 모든 해커들은 이런 종류의 성공을 얻기 위해 살아간다! 거기에는 내가 기준을 더 높이 잡아야 한다는 의미도 들어있다. fetchmail을 최상의 것으로 만들기 위해 나는 내자신의 필요뿐 아니라 나와는 상관없지만 다른 사람들에게는 필수적인 기능을 포함시키고 지원해야했다. 게다가 프로그램을 단순하고 튼튼하게 유지시키면서 그런 일을 해야 했다.

이것을 깨닫고 나서 내가 추가한 매우 중요한 첫 번째 기능은 멀티드롭(multidrop) 기능이었다. 그룹이나 사용자들의 메일을 한꺼번에 가지고 있는 메일박스에서 메일을 가져와 각 메일들을 개인 수신자에게 라우트(route) 시켜주는 기능이었다.

멀티드롭 기능을 추가하기로 한 데에는 몇몇 사용자들이 원한다는 것도 있었지만 가장 큰 이유는 어드레싱을 완전히 구현함으로써 싱글드롭 코드에 있는 버그들을 잡아낼 수 있으리라고 생각했기 때문이다. 그리고 그렇게 되었다. *RFC 822* `<http://info.internet.isi.edu:80/in-notes/rfc/files/rfc822.txt>`의 파싱을 제대로 구현하는 것에 매우 오랜 시간이 걸렸는데, 각각의 조각이 어려웠기 때문이 아니라 각각이 서로 의존하고 있으며 세심하게 신경을 써야 하는 사항들이었기 때문이다. 하지만 멀티드롭 어드레싱 역시 매우 훌륭한 설계상

의 결정이었던 것으로 드러났다. 다음과 같은 교훈을 얻을 수 있었다.

14. 어떤 도구든지 기대하는 방법으로 쓸모가 있어야 하지만 정말 위대한 도구는 사용자가 전혀 기대하지 않았던 용도에 알맞게 된다. (Any tool should be useful in the expected way, but a truly great tool lends itself to uses you never expected)

미처 생각하지 못했던 멀티드롭 fetchmail 의 용도는 메일링리스트를 그대로 유지한 채, 알리아스 확장이 된 채로 SLIP/PPP로 연결된 *클라이언트* 쪽에서 메일링리스트를 운영하는 것이었다. 개인의 컴퓨터로 ISP 계정을 통해 접속하는 사람이 ISP 의 알리아스 파일에 지속적으로 접근하지 않고도 메일링리스트를 운영할 수 있다는 것을 의미한다.

베타테스터들이 요구한 중요한 변경사항중 또 하나는 8비트 MIME 오퍼레이션이었다. 이것은 내가 코드를 8비트에 대비하여 계속 유지시켜왔기 때문에 매우 쉬운 일이었다. 이런 기능에 대한 요구를 미리 예측해서 그랬던 것은 아니다. 다음과 같은 규칙을 따르려고 해서였다.

*15. 어떤 종류든 게이트웨이 소프트웨어를 만들려고 한다면 데이터 스트림에 가능한 한 최소한의 조작만 가하라 – 그리고 수신자가 강제로 하게 하지 않는다면 정보를 *절대로* 잘라버리지 말라. (When writing gateway software of any kind, take pains to disturb the data stream as little as possible – and *never* throw away informtion unless the recipient forces you to!)*

이 규칙을 따르지 않았다면 8비트 MIME 지원은 매우 어려웠을 것이며 많은 버그를 만들어 냈으리라. 규칙을 따랐기 때문에 내가 해야 할 일은 *RFC 1652* `<http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1652.txt>` 를 읽고 헤더 생성 로직을 약간 수정하는 것 뿐이었다.

유럽의 몇몇 사용자들은 한 세션에서 가져올 수 있는 메시지의 수를 제한하도록 옵션을 추가해달라고 요구해왔다. (전화 네트워크의 비싼 비용을 조절할 수 있도록 해달라는 말이다) 오랫동안 여기에 저항했고, 아직도 완전히 수긍하지 못했다. 하지만 세계를 상대로 프로그램을 만든다면 고객들의 소리에 귀를 기울여야 한다 – 그들이 돈을 지불하지 않는다고 해도 마찬가지다.

제 8 절 Fetchmail에서 배울 점

일반적인 소프트웨어공학의 주제로 돌아가기 전에 fetchmail 의 경험으로부터 배울 점이 몇 가지 더 있다. rc 파일의 구문은 선택사항으로 'noise' 라는 키워드를 포함하는데 이것은 파서에 의해 무시된다. rc 파일에서 허용하는 영어와 비슷한 구문은 잘라낼 것을 모두 잘라낸 후에 얻는 전통적이고 간명한 키워드-밸류 짝에 비해 훨씬 알아보기 쉽다.

이것은 내가 rc 파일의 선언들이 명령형 소언어 (imperative minilanguage)를 얼마나 많이 닮아가기 시작했는지 알아차리고 나서 한밤중의 실험으로 시작되었다. (popclient 의 'server' 라는 키워드를 'poll' 로 바꾼 이유도 이것이다)

명령형 소언어를 더 영어처럼 만들면 사용하기 쉬울 것으로 보였다. 지금은 내가 비록 Emacs 나 HTML, 그리고 많은 데이터베이스 엔진에서 볼 수 있듯이 설계를 할 때 "언어처럼 만드는" 파의 일원이긴 하지만 "영어와 비슷한" 구분을 가지는 것에 대해서는 그다지 달가와 하지 않는다.

전통적인 프로그래머들은 정확하고 짧으며 중복을 허용하지 않는 제어구문을 선호하는 경향이 있다. 이것은 컴퓨팅 자원이 비싸서 파싱하는 단계가 최대한 싸고 간단해야 했을 때부터 내려온 문화적 유산이다. 영어는 대략 50% 정도의 중복을 허용하므로 대단히 부적절한 모델인 것으로 보인다.

이것이 내가 영어와 비슷한 구문을 일반적으로 피하는 이유는 아니다. 이 문제를 언급한 이유는 그런 관습을 없애기 위해서다. 사이클과 코어의 값이 싸졌는데도 간명함은 저절로 없어지지는 않았다. 최근에는 언어가 컴퓨터의 관점에서 싼 가격이라는 점보다는 사람에게 편리한가 하는 점이 더 중요하다.

물론 조심해야 할 이유는 충분히 있다. 한 가지는 파싱하는 단계의 복잡성에 대한 비용이다 – 파싱하는 단계를 버그가 우글거리는 데다가 사용자로 하여금 그 자체만으로 혼란을 일으키게 만들고 싶지는 않을 것이다. 또 하나의 이유는 언어의 구문을 영어와 비슷하게 만들려고 노력하면 그 "영어" 가 심각하게 왜곡되어 자연어와의 피상적인 유사점이 전통적인 구문만큼이나 혼란스럽게 되는 경우가 많다는 점이다. (소위 "4세대" 언어와 상

업용 데이터베이스 질의어에서 이런 경우를 많이 볼 수 있다)

fetchmail 제어구문은 이런 문제를 피하려고 했다. 언어의 영역이 매우 제한되어 있었기 때문이다. 일반적인 목적의 언어와는 거리가 멀었다. 언어가 표현하는 것이 별로 복잡하지 않았기 때문에 영어의 일부분에서 실제 제어언어로 옮겨가는데 혼란을 일으킬 가능성이 적었다. 더 넓은 의미의 교훈을 여기에서 얻었다.

16. 언어가 튜링-컴플리트하지 않다면 구문상의 유연성이 필요하다. (When your language is nowhere near Turing-complete, syntactic sugar can be your friend)

또하나의 교훈은 불투명함에 의한 보안에 대해서이다. fetchmail 의 사용자 중에는 스누퍼들이 우연히 패스워드를 보지 못하도록 rc 파일에 있는 패스워드를 암호화하여 가지고 있게 하자고 이야기하는 사람들이 있었다.

나는 그 이야기를 받아들이지 않았는데, 그렇게 한다고 해서 보안이 강화되는 것이 아니기 때문이다. rc 파일의 읽기 퍼미션을 얻은 사람이라면 사용자와 마찬가지로 fetchmail을 실행시킬 수도 있는 것이다 – 그리고 그들이 패스워드를 원하는 것이라면 패스워드를 얻기 위해 fetchmail 코드에서 디코딩하는 코드를 뽑아낼 수도 있다.

.fetchmailrc 의 패스워드를 암호화 했다면 사람들은 그리 심각하게 생각하지도 않고 보안에 대해 잘못된 관념을 가지게 되었을 것이다. 여기서 알 수 있는 일반적인 규칙은 다음과 같다.

17. 보안시스템은 그것이 보호하려고 하는 비밀만큼만 안전하다. 가짜 비밀들에 주의할 것. (A security system is only as secure as its secret. Beware of pseudo-secrets)

제 9 절 시장 스타일의 개발에 필요한 선행조건들

이 글을 초기에 검토해준 사람들과 시험적으로 청중이 되었던 사람들은 계속해서 성공적인 시장 스타일의 개발을 위한 선행조건이 무엇인지 물었다. 여기에는 공동 개발자의 공동체를 만들기 위해 프로젝트가 공개되는 시점에 리더의 자질과 코드가 어떤 상태인지가 포함된다.

아예 처음부터 시장 스타일로 개발할 수 없다는 것은 자명하다. 테스트, 디버그, 그리고 개선은 시장 스타일로 할 수 있다. 하지만 프로젝트를 *시작할때* 시장 스타일로 시작하기는 매우 어렵다. 리누스는 그렇게 하지 않았다. 나도 마찬가지. 개발자들의 공동체는 초기에 실행시키면서 테스트할 수 있는 장난감이 필요하다.

공동체를 만들기 시작할 때 제시할 수 있어야 하는 것은 *그럴듯한 장래성*이다. 프로그램이 특별히 잘 작동할 필요는 없다. 조잡하거나, 버그투성이여도 되고, 완성되지 않고 문서가 형편없어도 상관없다. 하지만 한가지 확실하게 해야할 것은 잠재적인 공동개발자들에게 이것이 머지 않은 미래에 정말 괜찮은 무언가로 진화할 수 있다는 것을 납득시키는 일이다.

리눅스와 fetchmail 둘 다 강력하고 매력적인 기본설계를 가지고 공개되었다. 내가 시장모델에 대해 이야기하자 많은 사람들이 이것을 중요하다고 생각했고, 높은 수준의 설계에 대한 직관과 영리함이 프로젝트의 리더에게는 필수적인 것이라고 지레짐작으로 결론을 내려버렸다.

하지만 리누스는 그의 설계를 유닉스에서 따왔고 나는 기본적으로 popclient에서 가져왔다. (물론 나중에 많은 것이 바뀌긴 했지만 리눅스는 그보다 훨씬 더 바뀌었다). 그렇다면 시장스타일의 리더/조정자에게 정말 특별한 설계의 재능이 필요한 것일까, 아니면 다른 사람들이 가진 설계의 재능을 이끌어 내는 것이 필요한 것일까?

나는 조정자가 특별하게 영리해서 독창적인 설계를 만들어낼 수 있는지의 여부가 중요하다고 생각하지 않는다. 하지만 조정자가 *다른사람의 좋은 설계를 알아볼 수 있는지* 는 절대적으로 중요하다.

리눅스와 fetchmail 프로젝트는 이에 대한 증거를 보여주고 있다. 리누스는 (앞서 논했듯이) 대단히 독창적인 설계자라고는 할 수 없으나 좋은 설계를 알아보는 대단한 요령을 보여주었고, 그것을 리눅스 커널에 통합해 넣었다. 앞서 fetchmail에서 가장 강력한 설계상의 아이디어 하나 (SMTP 포워딩)가 다른 누군가로부터 온 것이라고 설명한 바 있다.

이 글의 초기 청중들은 내가 설계상의 독창성을 과소평가하는데 내가 그런 독창성을 가지고 있기 때문에 당연한 일로 생각한다고 주장하며 내게 경의를 표시했다. 어느 정도는 사실이다. 설계는 분명히 (코딩이나 디버깅에 비해서) 내가 가장 잘하는 일이다.

하지만 소프트웨어 설계에 있어서 똑똑하고 독창적이라는 것의 문제점은 그게 버릇이 되어버린다는 점이다 –

설계를 강력하고 단순하게 유지해야 할 때 그렇게 하지 않고 계속해서 일을 멋지고 복잡하게 만들기 시작한다. 이전의 프로젝트에서 나는 그런 성향 때문에 실패한 적이 있었다. fetchmail 에서는 간신히 그것을 이겨냈다.

그래서 나는 fetchmail에서 성공할 수 있었던 것이 부분적으로는 똑똑해지려는 유혹을 이겨냈기 때문이라고 생각한다. 이것은 (최소한) 성공적인 시장 프로젝트에 설계상의 독창성이 필수적이라는 점에 대해서는 반대되는 주장이다. 리눅스를 생각해 보자. 리누스 토발즈가 개발도중에 운영체에 설계에 있어서 근본적인 혁신을 이끌어내려고 노력했다고 가정해 보자. 결과로 만들어진 커널이 우리에게 있는 것처럼 안정적이고 성공적이었을 것이라고 생각이나 할 수 있을까?

어느 정도 기본적인 수준의 설계와 코딩기술은 물론 필요하긴 하지만 시장 스타일의 프로젝트를 시작하려고 심각하게 생각하고 있는 사람이라면 그런 정도는 최소한 넘어섰으리라고 기대하는 것이다. 평판에 대한 오픈 소스 공동체의 내부시장은 미묘한 압력을 사람들에게 작용한다. 그래서 지속적으로 따라갈 수 있는 경쟁력을 가지고 있지 않은 사람이라면 개발 프로젝트를 시작하지 않게 된다. 지금까지 이것은 잘 들어맞아왔던 것 같다.

시장 스타일의 프로젝트에서 똑똑한 설계만큼이나 중요하다고 생각하는 것이지만 일반적으로 소프트웨어 개발과는 연관짓지 않는 또 한 종류의 기술이 있다 – 어쩌면 더 중요할 지도 모른다. 시장스타일의 프로젝트를 조정하거나 이끄는 사람은 사람들과 잘 의사소통하는 기술을 가지고 있어야 한다.

이것은 명확하다. 개발자들의 공동체를 만들려면 사람들을 끌어모아야 하고 무엇을 하고 있는지 그들에게 흥미를 주어야 하고 그들이 하는 일의 결과에 대해서 기분좋을 수 있도록 만들어 주어야 한다. 기술적인 논란(sizzle)은 이런 것을 이룩하는데 도움이 많이 되긴 하지만 그것이 전부는 아니다. 그 사람의 성격도 크게 작용을 한다.

리누스가 괜찮은 녀석이고 다른 사람들이 그를 좋아하게 되며 그를 도와주고 싶어한다는 것은 우연이 아니다. 내가 정력적이고 외향적이며 많은 사람들과 일하는 것을 즐기고 만화속의 인물 비슷한 인상을 주는 것은 우연이 아니다. 시장 모델이 성공하게 하려면 자신이 사람을 끄는 매력이 조금이라도 있는 것이 매우 큰 도움이 된다.

제 10 절 오픈 소스 소프트웨어의 사회적 문맥

다음과 같은 말이 있다. 가장 뛰어난 해킹은 해커의 일상적인 문제를 푸는 개인적인 해결책으로 시작한다. 그리고 그 문제가 많은 사용자들에게 전형적이라는 것이 밝혀지면 널리 퍼지게 된다. 첫번째 법칙으로 되돌아와 (아마도) 더 유용한 방식으로 다시 말해보자.

18. 재미있는 문제를 풀어보고 싶다면 자신에게 재미있는 문제를 찾아 나서는 것부터 시작하라. (To solve an interesting problem, start by finding a problem that is interesting to you)

칼 해리스와 popclient 가 그러했고, 나와 fetchmail 이 그러했다. 하지만 이것은 오래전부터 이해되고 있었다. 재미있는 점은 리눅스와 fetchmail 의 역사가 가리키고 있는 것처럼 다음 단계에 있다 – 사용자와 공동개발자가 이루는 크고 활동적인 공동체의 눈앞에서 소프트웨어가 진화해 가는 것이다.

"Man-Month 의 신화" 에서 프레드 브룩스는 프로그래머의 시간이 다른 것으로 대체될 수 없다고 진술했다. 지연되고 있는 소프트웨어 프로젝트에 개발자를 더 투입하는 것은 완료 시기를 더 늦출 뿐이다. 그는 프로젝트에서 복잡성과 의사소통에 드는 비용이 개발자의 제곱에 비례하는 반면 일이 되어가는 것은 직선적으로만 증가한다고 주장했다. 이 주장은 그때부터 "브룩스의 법칙" 으로 알려졌고 널리 자명한 이치로 간주되었다. 하지만 브룩스의 법칙이 전부라면 리눅스는 불가능했을 것이다.

나중에 제랄드 와인버그의 고전인 "컴퓨터 프로그래밍의 심리학 (The Psychology Of Computer Programming)" 에서 브룩스의 말에 대한 중요한 수정사항이 제시되었다는 것을 알 수 있다. "자아를 내세우지 않는 프로그래밍(egoless programming)" 에 대한 논의에서 와인버그는 개발자들이 자신의 코드에 대해서 텃세를 부리지 않고 다른 사람들로 하여금 버그를 찾고 개선가능성을 찾아내도록 격려하는 곳에서는 다른 어느 곳에서보다 극적으로 빠른 개선이 일어난다고 이야기했다.

아마도 와인버그의 분석이 적절한 평가를 받지 못했던 것은 용어선택의 문제 때문이었을 것이다 – 인터넷의 해커들이 "자아를 내세우지 않는다" 고 묘사하는 것에는 웃음을 지을 수밖에 없다. 하지만 나는 그의 주장이 지금 그 어느때보다 절실하다고 생각한다.

유닉스의 역사는 우리가 리눅스로부터 배우고 있는 것을 (그리고 내가 실험적으로 더 작은 스케일로 리누스의 방법을 따라 함으로써 검증한 것을) 미리 준비해 두었어야 했다. 다시 말해 코딩은 본질적으로 고독한 작업인데 비해 정말 중요한 해킹은 전체 공동체의 주의와 지력(brainpower)를 이용함으로써 이루어진다는 것이다. 폐쇄된 프로젝트에서 자신의 두뇌만을 사용하는 개발자는 수백명의 사람들이 버그를 찾아내고 개선을 이루어내는 열려있는 진화적 컨텍스트를 어떻게 만들어내는지 아는 개발자에게 뒤떨어지기 마련이다.

하지만 전통적인 유닉스 세계에는 이런 접근방법을 끝까지 밀어붙이지 못하도록 하는 요인이 몇 가지 있었다. 첫 번째는 다양한 라이센스의 법적인 제약, 거래 비밀, 그리고 상업적인 이해관계였다. 또 하나는 (나중에서야 알게 되었지만) 인터넷이 그리 훌륭하지 못했기 때문이다.

싼 가격에 인터넷을 이용할 수 있게 되기 전에는 지리적으로 좁은 지역에서 공동체가 자리잡고 있었고, 그 공동체의 문화는 와인버그의 "자아를 내세우지 않는" 프로그래밍이 장려되었으며 개발자들은 쉽게 많은 사람들, 숙련된 훈수꾼들과 공동개발자들을 끌어들일 수 있었다. 벨 연구소, MIT 인공지능 연구소, UC 버클리 – 이곳이 바로 전설적인 혁신들이 일어난 곳이고, 여전히 그런 잠재력을 가지고 있는 곳이다.

리눅스는 재능을 끌어올 풀(pool) 로 *전세계*를 사용하기 위해 의식적으로, 또 성공적으로 노력한 최초의 프로젝트였다. 나는 리눅스의 태동기가 월드와이드웹의 탄생과 일치하는 것을, 그리고 리눅스가 유아기를 벗어나던 1993-1994년 경에 ISP 산업과 인터넷에 주류의 관심이 폭발하기 시작했던 것을 우연이라고 생각하지 않는다. 리누스는 급속히 보급되는 인터넷을 가능하게 했던 그 규칙에 따라 어떻게 일을 진행해야 하는지 알았던 최초의 사람이다.

저렴하게 인터넷을 사용할 수 있었던 것이 리눅스 모델이 진화하는데 필수적인 조건이었긴 하지만 그것이 충분조건이라고 생각하지는 않는다. 또 하나의 중요한 요소는 리더쉽 스타일과 협력하는 관습의 발전인데, 이것이 개발자로 하여금 공동 개발자를 끌어모으고 매체를 최대한으로 활용하게 했던 것이다. 그렇다면 리더쉽 스타일이란 무엇이고 이런 관습이란 어떤 것인가? 권력관계에 기반한 것은 분명 아니다 – 만일 그런 것에 기반했다면 강제에 의한 지도력은 우리가 보고있는 것과 같은 결과를 내지 못했을 것이다. 와인버그는 19세기 러시아 무정부주의자인 표트르 알렉세이비치 크로포트킨의 자서전, "한 혁명가의 비망록" 으로부터 다음과 같은 구절을 인용하고 있다.

"농노를 소유한 가정에서 컸기 때문에 나는 내 시대의 모든 젊은이들처럼 능동적인 생활을 했다. 명령하고, 지시하고, 꾸중하고, 벌주는 그런 일에 대한 필요성을 크게 확신하고 있었다. 하지만 내가 큰 사업을 경영해야 했을 때는 다른 사람(자유인)들과 거래해야 했고 단 한 번의 실수가 심각한 결과를 가져올 수 있게 되었을 때 명령과 훈육의 원리에 기반해 행동하는 것과 공동이해의 원리에 의해 행동하는 것 사이의 차이점을 높이 평가하기 시작했다. 군대에서라면 전자에 의해 일하는 것이 훨씬 낫겠지만 실생활에서 많은 사람들의 의지를 수렴하여 노력해야만 이룰 수 있는 목표를 겨냥했을 때는 별 가치가 없다."

"많은 사람들의 의지를 수렴하여 노력하는 것" 이 바로 리눅스와 같은 프로젝트가 요구하는 것이다 – 그리고 "명령의 원리" 는 결과적으로 우리가 인터넷이라고 부르는 무정부주의자들의 천국에 사는 자원봉사자들에게 적용하기 불가능한 것이다. 효과적으로 일하고 경쟁하기 위해 공동 프로젝트를 이끌어보고 싶은 해커들은 크로포트킨의 "이해의 원리" 가 어렴풋이 제시하고 있는 방식에 따라 같은 관심을 가진 공동체를 어떻게 효과적으로 끌어모으고 격려할 것인지 배워야 한다.

앞에서 나는 리누스의 법칙을 설명하기 위해 "델파이 효과"를 언급했다. 하지만 생물학과 경제학에서의 적응계에 비유하는 것이 더 강력한 비유라고 할 수 있다. 리눅스 세계는 많은 점에서 생태계나 자유시장과 같이 행동한다. 일단의 이기적인 에이전트들이 효용을 극대화시키기 위해 애쓰는 과정을 통해 스스로를 수정하는 자율적인 질서를 만들어 내며 이것은 중앙통제가 이룰 수 있는 어떤 결과보다 더 정교하고 효율적이다. 그렇다면 여기에서 "이해의 원리"를 찾아낼 수 있다.

리눅스의 해커들이 최대화하려고 하는 "효용함수" 는 고전적인 의미에서의 경제적인 것은 아니고 그들 자신의 측정할 수 없는 자아 만족과 다른 해커들 사이의 평판이다. (이런 동기를 "이타적" 이라고 말할 지도 모르겠지만 그렇게 말하는 것은 이타주의 그 자체가 이타주의자의 자아를 만족시키는 한 형태라는 사실을 무시하는 것이다) 이런 방식으로 일을 처리하는 자발적인 문화는 사실 그렇게 찾아보기 힘든 것은 아니다. 내가 오랫동안 참여해왔던 또 하나의 문화는 과학소설 팬들의 세계 (science fiction fandom) 이다. 해커들의 세계와 다르지 않게 여기에서는 "자아상승" (다른 팬들 사이에서 자신의 평판이 높아지는 것) 이 자발적인 활동 뒤의 기본적인 동기라고 분명하게 인식한다.

리누스는 자신을 개발은 대부분 다른 사람들에 의해 이루어지는 프로젝트의 수문장으로 위치시키는데 성공했고, 프로젝트가 스스로 유지할 수 있게 될 때까지 계속해서 흥미거리를 공급해 줌으로써 크로포트킨의 "공유이해의 원리" 의 의미를 정확하게 따랐다. 준-경제학적인 관점에서 리눅스 세계를 보면 어떻게 이해가 적용되었는지를 알 수 있다.

리누스의 방법을 "자아 상승" 에 있어서 효과적인 시장을 만드는 길로 볼 수 있다 – 개개인인 해커들의 이기심을 최대한 단단하게 지속적인 협동으로만 이룩할 수 있는 어려운 목적과 연결시키는 것이다. fetchmail 프로젝트에서 나는 (더 작은 규모였지만) 이 방법을 그대로 따라했고 좋은 결과를 냈다. 아마도 내가 리누스보다 더 의식적이고 체계적으로 일을 해냈을 것이다.

많은 사람들이 (특히 자유시장을 정치적으로 믿지 않는 사람들) 스스로에게 방향이 맞추어진 이기주의자들의 문화가 파편화되어 있으며 텃세가 심하고 소모적이면서, 비밀이 많고 적대적일 것이라고 생각한다. 하지만 이런 기대는 (예를 하나만 들자면) 리눅스 문서의 놀랄만한 다양성과 질, 깊이에 의해 산산이 부서지고 만다. 프로그래머들이 문서작업을 *끔직하게 싫어한다*는 것은 모두가 기정사실로 받아들이고 있다. 그렇다면 대체, 리눅스 해커들이 문서를 그렇게 많이 만들어냈다는 것은 어떻게 설명할 것인가? 분명히 리눅스의 자아상승을 위한 자유시장은 막대한 자금이 들어간 상업용 소프트웨어 프로듀서들의 문서작업보다 다른 사람을 위한 고결한 행동을 더 잘 해낸 것이다.

fetchmail 과 리눅스 커널 프로젝트는 둘 다 많은 해커들의 자아를 적절히 보상해 줌으로써 강력한 개발자/조정자 가 인터넷을 이용하여 많은 수의 공동개발자를 가지는 이익을 얻으면서 프로젝트가 혼돈스럽게 스스로 붕괴하는 것을 막을 수 있다는 것을 보여준다. 브룩스의 법칙에 대해서 나는 다음과 같은 반대제안을 한다.

19. 개발 조정자가 최소한 인터넷만큼 좋은 매체를 가지고 있으며 강제력을 사용하지 않고 어떻게 이끌어야 할지 알고 있다면 한 명 보다는 여러명의 리더가 필연적으로 더 낫다. (Provided the development coordinator has a medium at least as good as the Internet, and know how to lead without coercion, many heads are inevitable better than one)

미래의 오픈 소스 소프트웨어는 점점 리누스의 게임을 어떻게 해야 하는지 아는 사람들, 성당을 뒤로 하고 시장을 끌어안을 수 있는 사람들에게 속할 것이라고 생각한다. 개인의 비전과 똑똑함이 문제가 되지 않으리라는 말이라기보다는 오픈 소스 소프트웨어의 최첨단은 개인의 비전과 똑똑함에서 시작하여 자발적으로 흥미를 보이는 공동체를 효과적으로 구축해서 그것을 증폭시키는 사람들에게 속할 것이라는 뜻이다.

그리고 그것은 오픈소스 소프트웨어의 미래에만 국한되지는 않을 것이다. 닫혀있는 소스로 개발하는 사람은 리눅스 공동체가 문제를 해결하기 위해 끌어낼 수 있는 재능의 풀과 경쟁할 수 없다. 극소수만이 fetchmail 에 공헌했던 200명보다 많은 사람을 고용할 수 있을 것이다. 아마 최종적으로는 협동이 더 도덕적이라거나 소프트웨어 "매점" 이 덜 도덕적이라서가 아니라 단지 닫혀진 소스 측과 오픈 소스 공동체와의 군비경쟁에서 오픈 소스 측이 한 문제에 훨씬 큰 비율로 숙련된 사람의 시간을 쏟을 수 있기 때문에 오픈 소스 문화가 승리를 거둘 것이라는 얘기다.

제 11 절 감사의 글

이 글은 많은 사람들과의 대화를 통해 잘못을 수정하는데 도움을 받았다. 특별히 제프 덧키(Jeff Dutky : dutky@wam.umd.edu) 에게 감사한다. 그는 "디버깅은 병렬처리가 가능하다" 는 말을 제안해 주었고 그로부터 이어지는 분석을 발전시키는데 도움을 주었다. 낸시 레보비츠(Nancy Lebovitz : nancyl@univers.digex.net) 에게도 감사한다. 그녀는 크로포트킨을 인용하여 내가 와인버그를 흉내내도록 도와주었다. General Technics 리스트의 조안 에슬링거 (Joan Eslinger wombat@kilimanjaro.engr.sgi.com) 와 마티 프란츠 (Marty Franz marty@net-link.net) 도 예리한 비판을 보내주었다. 폴 에거트 (Paul Eggert eggert@twinsun.com) 는 GPL 과 시장 모델의 상충되는 점을 알려주었다. 이 글의 첫 번째 공개버전의 첫 번째 시험적인 청중이 되어준 필라델피아 리눅스 사용자 모임 (PLUG : Philadelphia LInux User's Group) 의 멤버들에게 감사한다. 마지막으로 리누스 토발즈의 논평은 도움이 되었으며 초기에 그가 해준 추천은 매우 격려가 되는 것이었다.

제 12 절 읽어볼 만한 글들

프레드릭 브룩스 (Frederick P. Brooks) 의 고전인 *[Man-Month 의 신화 (The Mythical Man-Month)]*에서 몇몇 부분을 인용했다. 앞으로도 여러 관점에서 그의 통찰력을 발전시킬 수 있을 것이다. 애디슨-웨슬리(Addson-Wesley) 의 25주년 기념판 (ISBN 0-201-83595-9)을 추천한다.여기에는 그가 1986년에 쓴 글, "*은총알은 없다 (No Silver Bullet)*" 가 들어있다. 새 기념판은 매우 귀중한 20년 후의 회고를 담고 있다. 브룩스는 여기서 원문의 몇몇 판단이 시간이 흐름에 따라 옳지 않은 것으로 드러났다고 솔직하게 인정하고 있다. 나는 이 글을 대략 마무리 지은 후에 회고담을 읽어보았는데, 브룩스가 시장 스타일을 마이크로소프트에서 연유한 관습으로 생각한다는 것을 발견하고 깜짝 놀랐다!

제랄드 M. 와인버그의 *[컴퓨터 프로그래밍의 심리학 (The Psychology Of Computer Programming)]* (New York, Van Nostrand Reinhold 1971) 은 비운의 개념인 "자아를 내세우지 않는 프로그래밍" 을 소개했다. "명령의 원칙" 이 무용지물이라는 것을 처음으로 깨달은 사람이 와인버그는 아니지만 그는 아마도 처음으로 그것을 인식하고 특별히 소프트웨어 개발과 관련하여 논지를 전개시킨 첫 번째 사람일 것이다.

리차드 P. 가브리엘 (Richard P. Gabriel) 은 리눅스 이전 시대의 유닉스 문화에 대해 숙고하고 주저하면서도 원시적인 시장 스타일의 모델이 우월하다는 것을 1989년의 글인 *[리스프 : 좋은 소식과 나쁜 소식, 그리고 큰 성공을 거두는 방법 (Lisp : Good News, Bad News, and How To Win Big)]* 에서 밝혔다. 몇 가지 시대에 뒤떨어진 감은 있지만 이 글은 여전히 리스프의 팬 (나를 포함해서) 들에게 적절한 찬사를 받고 있다. 편지를 교환하던 사람 중 한 명이 나에게 "나쁜 것이 좋은 것이다 (Worse is Better)" 라는 제목을 가진 절이 리눅스를 예견하다시피 했다는 것을 상기시켜 주었다. 이 글은 웹에서 다음의 주소에 가면 읽을 수 있다. `<http://www.naggum.no/worse-is-better.html>`.

드 마르코 (De Marco) 와 리스터 (Lister) 의 *[피플웨어: 생산적인 프로젝트와 팀 (Peopleware : Productive Projects and Teams)]* (New York; Dorset House, 1987; ISBN 0-932633-05-6) 은 결코 평가절하될 수 없는 보석이다. 프레드 브룩스가 회고의 글에서 이 책을 인용해서 기뻤다. 저자가 말하고 있는 것중에서 리눅스나 오픈 소스 공동체에 바로 적용될 수 있는 것은 거의 없지만 창조적인 작업의 필요조건에 대한 통찰력은 날카롭고 시잘 모델의 미덕을 상업적인 문맥에 결합시키려고 하는 사람에게는 가치있는 것이다.

마지막으로 사실 나는 이 글을 "성당과 광장 (The Cathedral and the Agora)" 이라고 이름붙이려고 했다. 광장은 그리스어로 열린 시장이나 공개 집회장을 뜻한다. 마크 밀러 (Mark Miller) 와 에릭 드렉슬러 (Eric Drexler) 의 생산적인 글, "광장 시스템" 에 관한 논문들은 계량 생태학과 비슷한 시장의 속성들을 묘사함으로써 5년 후에 리눅스를 알게 되었을 때 오픈 소스 문화에서 일어나는 현상들을 그에 비유하여 생각할 수 있도록 도와주었다. 이 글들은 웹의 다음 주소에서 구할 수 있다. `<http://www.agorics.com/agorpapers.html>`.

제 13 절 후기 : 넷스케이프가 시장 스타일을 받아들이다!

역사가 만들어지는 데 일조를 했다는 사실을 깨닫는 것은 좀 이상한 느낌이다.

1998년 1월 22일, 내가 처음으로 이 글을 발표한 지 7달 정도 지난 시점에서 넷스케이프 사가 *넷스케이프 커뮤니케이터의 소스를 공개* `<http://www.netscape.com/newsref/pr/newsrelease558.html>`하기로 했다고 발표했다.

나는 발표가 있기 전날 까지도 이런 일이 일어나리라고는 생각하지 못했다. 넷스케이프의 수석 기술담당이자 부사장인 에릭 한(Eric Hahn) 은 발표직후에 다음과 같은 email을 보냈다. "넷스케이프의 모든 사람들을 대신해 우리가 이곳까지 오도록 도와주신 것에 감사드립니다. 당신의 생각과 글이 우리의 결정에 근본적인 영감을 주었습니다."

그 다음주에 나는 넷스케이프 사의 초청으로 실리콘 밸리에 가서 고위 경영진 및 기술진들과 함께 하루짜리 전략회의(1998년 2월 4일)에 참석했다. 우리는 넷스케이프의 소스 공개 전략과 라이센스를 함께 설계했고 최종적으로는 오픈 소스 공동체에 크고 긍정적인 영향을 끼칠 것으로 희망하는 몇몇 계획을 만들었다. 이 글을 쓰는 시점에서 더 자세히 언급하기에는 너무 이르지만 수주일 내에 자세한 사항이 발표될 것이다.

넷스케이프는 시장모델을 상업계에서 대규모로 실제 테스트할 수 있는 기회를 제공하려 한다. 오픈 소스 문화

는 이제 위험을 맞이하게 된 것이다. 넷스케이프의 시도가 실패한다면 오픈 소스 개념은 불신을 받을 것이고 상업계에서 향후 십년간은 오픈 소스를 다시 받아들이려 하지 않을 것이다.

반면에 볼만한 기회가 될 수도 있다. 월스트리트 등의 첫 반응은 조심스럽지만 긍정적이었다. 우리 자신을 증명할 기회를 얻은 것이다. 만일 넷스케이프가 이번 행보로 상당한 양의 시장점유율을 끌어올린다면 컴퓨터 산업에서 오래 전에 이루어졌어야 했던 혁명을 시작하게 되는 것이다. 다음 한 해는 매우 교육적이며 재미있는 한 해가 될 것이다.

제 14 절 버전과 변경 이력

$Id: cathedral-bazaar.sgml,v 1.38 1998/05/13 17:29:31 esr Exp $

1997년 5월 21일, 리눅스 회의(Linux Kongress)에서 1.16 버전을 발표

1997년 7월 7일, 1.20 버전에 문헌목록을 추가

1997년 11월 18일, 1.27 버전에 Perl 컨퍼런스에서 있었던 일화를 추가

1998년 2월 9일, 1.29 버전에서 "프리소프트웨어"를 "오픈 소스" 로 변경< 1998년 2월 10일, 1.31 버전에서 "후기: 넷스케이프가 시장스타일을 받아들이다"를 추가

그 외의 리비전(revision) 은 간단한 편집 및 마크업 수정임