

Synchronization of Distributed Development Software

Arthur C. Norman* & Anthony C. Hearn†

*Trinity College Cambridge CB2 1TQ, England

acn1@cam.ac.uk

and

†RAND, Santa Monica CA 90407-2138, USA

hearn@rand.org

Abstract

The REDUCE development and support effort is spread across several countries, and some of the people involved do most of their REDUCE work on machines that do not have high speed, reliable and cheap Internet access — sometimes computers at their homes that use dial-up links. Exploiting REDUCE's portability, developers use many different kinds of computer, and in particular some use Unix systems while others are based on Windows. At times Macintoshes and other systems have also been relevant. There are two different Lisp systems used, which give different trade-offs between speed, space and portability and so suit different circumstances. Given this diversity it has not always been easy in the past to achieve the apparently simple ideal of making it easy for all the developers to be assured that they have a fully up-to-date set of files on their machines. Experience over numbers of years have repeatedly uncovered new failure-modes for system distribution and we have been regularly reminded that the ideal of fast and reliable networks can not be assumed available across a sufficiently diverse community of users. We have therefore developed tools that use the Internet to allow remote sites to keep in step with the various places where master copies of parts of REDUCE-related material live. We concentrate on using the HTTP protocol, since support for that, both on servers and clients, is most generally available and least disrupted by firewalls and other security measures. Part of the intent of the procedures we use is that updating can usually happen at a fairly fine level of granularity. If an attempt to update a local copy is disrupted by temporary network failure, a second attempt will smoothly continue from where the previous try failed. As well as automating the fetching of data we have associated scripts that perform minimal reconstruction of local REDUCE binaries to account for whatever changes have been downloaded. Thus from the user's perspective the whole process of updating their system becomes very simple indeed. The procedure also supports the initial installation of a REDUCE development tree at a fresh site — a small set of management tools has to be fetched and installed manually, but after that they will “update” an initially empty filesystem until the whole system is fully in place.

1 Introduction

The work reported here represents one of a number of ways in which the REDUCE algebra system is evolving to take advantage of the recent explosive growth of the World Wide Web. Although fuller details of other Web-related REDUCE projects will appear elsewhere, we present a brief outline of some of them here to show how we are exploring ways of making all aspects of REDUCE's development, distribution, support and use fit into emerging net-

work paradigms. Our first concern has been with using the Web to distribute and update source code in the form required by developers and testers of the system. Concentrating on the REDUCE developers keeps the community involved in our experiments reasonably small initially, and keeping their source code in step avoids at least some of the problems associated with the wide range of computer architectures on which REDUCE runs. Previous ways of organizing the REDUCE collaboration had had difficulty supporting group members whose active involvement was irregular, or who did not have fast and reliable network links to the main computer used in this work, and even-handed support for multiple operating systems proved hard to achieve. This paper reports on the organizational disciplines and software tools that we have developed to get around these problems. An explicit part of our planning has been based on an expectation that in the near future most software will be distributed over networks, and users will expect upgrades and bug fixes to be fetched and installed for them in an almost transparent manner. We expect that the methodology used to support REDUCE developers today will grow into a fuller distribution and support scheme tomorrow. This will necessarily increase the proportion of participants in our scheme whose network connections involve low bandwidth links. At the other extreme we expect that prime collaborators with fast network access will increasingly find system managers in their institutions installing more and more elaborate firewall facilities and limiting full access to their systems to just locally registered users to improve institutional security.

As a possible additional component in this strategy, one of the Lisp systems use to support REDUCE[7] provides an architecture independent byte-code representation for REDUCE programs, and a project[1] is under way to support fetching such compiled code across network links and installing it directly into a running copy of REDUCE. In addition, it will provide a “just-in time” compiler (c.f. Kaffe¹, which does a similar job for Java) that will validate downloaded bytecode streams and then compile them into native code for the computer being used. Even if such a just-in-time compiler is not provided for every architecture that REDUCE runs on, overall system functionality will not be lost since bytetimes can always be interpreted (as they are at present).

In this paper we concentrate on the support of the distributed development of REDUCE. An initial and obvious concern is to ensure that each participating site is able to maintain an up-to-date mirror of the master REDUCE development tree. This apparently trivial problem is one that has caused an unexpected amount of practical difficulty over the years. To clarify this, we shall therefore start by sketching the history of the distribution mechanisms that have been used with REDUCE and documenting some of the frustration that arose with earlier schemes.

¹<http://www.tjwassoc.demon.co.uk/kaffe/kaffe.htm>

2 Previous Methods for Updating REDUCE

Since its first distribution in 1968, most users of REDUCE have received their initial copy of REDUCE on tape, floppy disks or other transportable media. Updates have then made available at infrequent intervals, usually about every eighteen months. In between full releases numerous changes to the code will be made, reflecting additional features, new packages, and inevitably, bug fixes. Over the years, various mechanisms have been used to bridge the gap between releases so that users could have relatively up-to-date software. This is an area where computer networking proved to be an important tool. An obvious candidate in this regard was FTP. However, when our distributed development project first begun, most of the REDUCE user community did not have access to this service, since many were not directly connected to the Internet or its predecessors. Specifically in a number of institutions a single special machine was provided with external network access but filespace and computing power on that system could be seriously constrained. As a result, changes were initially distributed where possible by e-mail, which could more often be delivered directly to the main machine one of the collaborators used. The Netlib system[2], which automatically processes e-mail requests that arrive at a server and replies to the sender with a copy of the requested file, was a distinct improvement in this regard. This enabled the REDUCE community over the past decade to obtain updates when they wanted them, rather than depend on automatic mailings that were hard to track.

The advent of gopher servers on the Internet provided an even more convenient means of access than Netlib, since it was much easier to browse the available material and download any needed items. Again, like many others, the REDUCE community quickly made use of this facility.

In spite of these advances, system maintenance was still a fairly difficult task. A user picking up a new or updated package had to install it by running a script after installing the obtained files in the appropriate places. In addition, it was not possible to put complete versions of the core system packages on a publicly accessible server and still maintain control over the distribution of the software. To resolve this problem, we developed a “patch” protocol a few years ago that enabled individual procedures anywhere in the system to be redefined in a single file. Users could either receive updated versions of this file by e-mail or collect it from the gopher server, and by running a script they could then bring their executable files up to date. Since the updating was based on the original released version, there was no problem with synchronization of code and little need to worry about who could gain access to the patches. While for some users this worked well, the need to carry out several manual steps to install the update resulted in many users not taking advantage of the capability.

Independent of this general distribution, another version of REDUCE had been made available to a small group intimately connected with its development. This *development version* was structured differently from the standard distribution. In the latter, the source code was organized in files that define *packages*. These encompassed a relatively complete capability, such as integration, or the solution of various classes of equations. Each file contained smaller units called *modules* that define sub-tasks within that capability. For the release version this large-granularity structure helped REDUCE run on (now historical) systems that lacked hierarchical file systems or which had severe limits on the number of files that could exist in any one directory. In the development version, the modules are themselves independent files, and each package is thus naturally stored in a separate directory. To facilitate their maintenance, each module is constructed so that it is independently compilable. Necessary declarations, macros and so on, that apply across the whole package, are contained in a module with the

same name as the package. For example, the `solve` package contains eight modules. As we just mentioned, the first has the same name (`solve`) as the package. The other seven modules (`solve1`, `ppsoln`, `solve1nr`, `glsolve`, `solvealg`, `solvetag`, and `quartic`) each take care of a particular case in the overall solving process. In general, such modules are a few hundred lines of code, and compile into the order of a few tens of kilobytes of object code. As a result, even on a loaded network, it is a fairly efficient process to download code for an individual module from a server, whereas a complete package requires much more time.

To keep the various copies of this software in step, developers sent their updated source code to a central repository, maintained at RAND. Once a week, a Unix `shar` file was produced from this repository containing all files that had changed or been created since the previous `shar`, and e-mailed to members of the development group. Those individuals were then expected to install these changed files on their machines, compile them, and then rebuild the executable image if any code in that image had changed. Although some steps of this process could be automated, it still required some work to keep a system up-to-date. Also, if one did not install every update, the system was not longer compatible with the code used by the rest of the community. Finally, the protocol did not provide for the *deletion* of files as opposed to their creation or updating, and required special handling for binary files, since they needed to be encoded for e-mail transmission.

This protocol worked well for those developers who received their mail on a Unix system and built and tested REDUCE on the same machine, and where their work on REDUCE was a central and consistent part of their main week-by-week work. However, it was much less comfortable for people whose work on REDUCE was more sporadic, or who did that work on less well-connected machines. And against all reasonable expectations even into 1999 there has been consistent experience of unreliable e-mail communication between some pairs of relevant sites.

When the World Wide Web burst on the scene, we therefore decided to design a new model for the distribution of this software, which, by capitalizing on the Web’s functionality, could address the problems in the process described above: specifically it could be based on “pull” rather than “push” technology, it would allow for unreliable links by validating the complete REDUCE fileset and supporting retries after any network or other failure. It would cope with file deletion and directory re-structuring as well as just simple updates, and the entire process should be as automated as possible across multiple platforms.

3 Structure of the New Model

The techniques we describe here are more general than the specific case of maintaining the REDUCE development system. They are intended to make it easier to keep sets of files at several sites in step, while keeping the volume of traffic between them under control. It also gives the remote sites the opportunity to initiate the process of getting themselves into exact agreement with the master site regardless of how long it may have been since they last did so. Thus a developer can leave the local copy of the REDUCE system static for a while, providing complete stability while some new experimental piece of code is tested, and then get back into step again when the time comes for integrating their new code with the full system. Mirror sites of software repositories solve a similar problem by using time-stamps to track when files change, and FTP as a transport mechanism. We had however experience over several years of situations where fragments of REDUCE code were shipped across routes that involved several intermediate computers and eventual transfer on floppy discs. That combined with individual machines

with unreliably set clocks and time-zone muddles had left us disenchanted with solutions based solely on time-stamps: they can get lost or corrupted too easily. We also considered the use of FTP unsatisfactory. There were two reasons for this: the first was (once again) the problem of machines not fully connected to the main international networks, while the second was because both of our institutions (and in fact many others world-wide) are configured with quite elaborate firewall systems that mean that anonymous FTP areas are frowned upon by system managers, while creating suitable non-anonymous accounts for collaborators can be even harder. Schemes that would be ideal for use by the managers of archive sites did not fit the flexible, informal and changing structure needed to support a modest size research and development group.

We arranged the files to be distributed so they all lived together in one (or a small number of) directories together with the associated sub-directories. A few extra control files then live in the root of such a directory tree and specify options and record status information. Different sites will hold master copies of various of these directories, and the objective is that one or more slave sites can be assured of having an exact copy of the complete set. Normally, sending updates involves a transaction initiated at the slave site (i.e., a “pull” process), and in its simplest form this will only involve read operations on Web pages published by the master sites. At present for this sort of development project we are content to rely on either HTTP’s basic authentication (which uses passwords but does not provide high security) or web server facilities that restrict access to pages to sites whose addresses are listed in a server configuration file. It is clear that more sensitive code would protecting with encrypted links; we see the problems involved as more in politics and regulations than in the technology.

Master sites need to run periodic maintenance tasks to keep index and status information on their servers up to date, but the whole scheme involves the least possible conflict with institutional security policies. Of course in cases where partners are able to establish suitable local accounts it becomes possible for a master site to initiate the updating process.

The update process is intended to support multiple platforms, and, in the extreme case, be completely platform independent. This means we are not limiting ourselves to Unix workstations operating over reliable, high-bandwidth, local networks. In particular, the model must be able to handle Macintosh, MS-DOS, and Windows file conventions as well as Unix, and will accept that transfers may be limited to exchanges over noisy channels of low bandwidth and high latency that are subject to interruption, including dialup and wireless links. Once again we are very aware of both the ideal of high-reliability high-bandwidth networks, and of the best case performance we have experienced, but we have also observed that both of our institution’s servers are occasionally off-line without us having seen advance notice of this, and that both national and international networks can sometimes saturate to a level where fetching even a few kilobytes of material is difficult. As well as allowing for ways in which current academic network connections do not always live up to their ideal, we were keen to look forward to the support of increasing numbers of people interested in REDUCE but linking to update services via dial-up links from their homes or offices.

The directory structure of the REDUCE development tree reflects the various components of the software distribution, such as source, documentation, test files, and so on. Since REDUCE is based on Lisp, the necessary Lisp support is also included in another sub-directory. One special feature is that two different Lisps can be used, namely CSL (Codemist Standard Lisp)[7] and PSL (Portable Standard Lisp)[4]. Both of these Lisps are based on the Standard Lisp model[5] first introduced (mainly for the support of REDUCE) nearly thirty years ago. In addition, the Lisp support for

execution varies from platform to platform. As a result, the structure of the *lisp* sub-directory reflects both Lisp and platform type. Since each separate Lisp/platform pair can be kept in a separate directory, it is possible to use the same overall REDUCE tree for any supported option. In other words, once common files have been downloaded, it is only necessary to download the relevant Lisp code to support a new platform. The two different Lisp systems provide a useful illustration of the need for master versions of files to live in a number of different places: it is both natural for the prime support for each Lisp system to be conducted by a different group from that which looks after the core of the algebra system. When executable binaries are provided for several different machine architectures these will most naturally be built and hence mastered on different computers. The same issues arise with some of the individual REDUCE algebraic packages.

4 Structure of the Support Code

Our support code has two major parts. The first is concerned with fetching files and directories, the second with performing minimal re-compilation to REDUCE when parts of its source change.

We started with code to fetch a file from a Web or FTP server. A Perl script by J. Friedl called *webget*² was selected for this purpose. Later on, to meet our particular needs, we implemented an adaptation of this in C. (There are other utilities with similar capabilities, such as *wget*³, which is also written in C. Regardless of the language they are implemented in, versions using the HTTP and FTP protocols are not especially hard to implement).

As mentioned earlier, we were not comfortable with using time-stamps to track when files changed. Updates based on just version number information would not protect against accidental corruption of files or ill-disciplined editing. We therefore concluded that we should identify versions of files using checksums. Thus we needed the capability, which has been encapsulated in a C utility *xport*⁴, to build a list that contains the name of each file and directory at any level in the tree in which it is called, together with its checksum. This list is stored in a control file *xport.chk*. Despite not using time-stamps to control downloading we also record them in *xport.chk* so we can give a slave file the same time-stamp as the master file. A configuration file can be used to indicate which files should be processed in text and which in binary mode and to allow mixed case filenames to be published even on platforms where the basic file system is mono-case. To cope with large filesets the catalog information described above is stored as a base version (which should change only rarely) and what is expected to be a much smaller file of increments, so to check status a remote site normally just needs to fetch this small increment report.

There is support for connecting to a remote master site and interrogating a file list there, and then either reporting how the local directory differs from the master one or downloading files to bring the relevant files at the slave site in synch with those at the master site. To use the “fetch” mode, *xport* needs to know the URL of the master site. It is also convenient to direct it to ignore files with certain classes of names (e.g. those associated with backup copies of edited files). The owner of a master site places this information, together with various other administrative directives, in a file called *xport.pat*.

The *xport* utility can compute checksums and reset time-stamps while fetching files, but for testing purposes we also needed stand-alone programs (*checksum* and *tstamp*) to perform these

²<http://www.cpam.org/modules/by-authors/JeffreyFriedl>

³<http://sunsite.auc.dk/ftp/pub/infosystems/wget/>

⁴<http://gauguin.trin.cam.ac.uk/util/xport.exe>

operations. Of course, had we only been concerned with Unix systems we would have been able to use Unix-standard utilities for these — but again our desire for a platform independent solution led us to collect or build our own set of tools.

The above utilities, being coded in C, need to be distributed manually to each participant in our collaboration. In most cases it will be easy to provide pre-compiled binaries. However, the code has been written in a conservative manner, so we expect that its adaptation to any new platforms (provided they support network access) will not be a problem. If we were starting the project now, we might have considered using Java rather than C to give us portable code well-suited for network applications.

With the above utilities available, the code that arranged to re-build REDUCE could be written as a relatively straightforward Perl script. This first checks that the Lisp, platform type, and the location of the REDUCE directory have been passed as arguments, or otherwise prompts the user. It then downloads all necessary files by iterating through a list of URLs defining where the relevant files are, starting with the utilities described earlier in this section. These URL-based filesets are mutually exclusive, so there is no problem with the order in which the files are loaded. Information defining the relevant filesets is contained in the `xport`-produced files included in the directory defined by each URL. By comparing a locally constructed `xport.chk` file with the version fetched from a master site it is easy to identify the exact set of files that have changed and hence only fetch the minimum amount of information that is needed.

Early versions of this script worked well when the Internet operated without error. However, we quickly discovered that even in ideal circumstances (where master and slave machines were at the same site), network delays could be sufficiently severe that the fetch of a file was aborted due to time-outs. Consequently, it was necessary to modify the code so that it recovered more robustly from such timeouts. It was important to recognize though that, apart from a catastrophic problem such as a failure to obtain say the checksum file, the download could continue on other files even when one failed. Once all files had been checked, the script then aborted if any downloading failures occurred, requiring the user to run it again to complete the installation. It would clearly be feasible to make the script automate periodic retries of fetches that fail, but at present we consider it better to make the user do this manually by starting the whole script afresh.

Once the script has determined that all files in the slave system match those on the masters, it builds the REDUCE executable. For CSL, this requires compiling an ANSI C-based Lisp kernel (at least if any of the C sources of CSL have been changed), and then letting CSL itself compile the REDUCE code. For PSL, a basic set of executables for the given machine platform are placed on the server, but again once those have been downloaded, individual REDUCE modules can be re-compiled locally. Users have had little trouble with these steps, since they were all starting from the same fileset, which presumably had been checked at the master site for integrity.

5 Security and Reliability

Initially, we controlled access to REDUCE development material by using one of the standard control methods present in our Web servers, namely that access to the relevant documents was only possible from client machines that have been registered. Depending on the exact way a Web server has been set up, changing such permissions may be as simple as altering a configuration file in a user's own filesystem. However, even if centrally managed configuration files need adjusting, we have found that this is handled smoothly by our administrations. The degree of protection that is supported

in this way is, of course, rather weak. However if we were seriously concerned we could arrange that we only placed encrypted files on the servers and that we then only provided the associated decryption keys to members of our development group. We intend, in the future, to extend the `xport` utility so that as well as updating a mirror site in its current plain-text form it can be used by a publisher to bring a publically accessible web site up to date with the encrypted forms of a set of files, and to allow a remote user who has the correct key to fetch from this site and find the decoded versions installed on their machine.

One problem with using an access method based on registered machines is that the existence of firewalls and proxy servers, plus the fact that most of our developers use a variety of machines, can cause access problems. The alternative of controlling access by means of an id/password pair is in fact easier to administer although it may be less secure. We are now testing this method as well.

A second security concern is that of files damaged in transit, and hence able to wreck REDUCE if installed into a previously working system. We rely on our checksums to provide verification of correct and complete downloading as well as to identify when downloading is needed at all.

A final security issue involves the hypothetical case of a rogue site setting up as an apparent REDUCE master site and spreading unwanted or unauthorized code across the Web. Several of these concerns could be addressed by using cryptographically strong checksums and by applying digital signatures to at least our `xport.chk` files, but to date we have not felt the need to apply such careful controls. We are confident that as such issues become important to development groups such as our own, it will be possible to exploit some of the high quality Web security mechanisms that are being developed to support electronic commerce. These should provide us with all the tools we need to control both access to and authentication of all the code we distribute.

6 Relationship to previous projects

There have been several systems in the past built for network distribution of software. One of the most well-known is the `rdist` system available as part of the Berkeley Unix distribution. `Rdist` “pushes” files to its clients, usually on a regular schedule in order to keep files on a local network in synch. An approach closer to ours is used by `coda`⁵, in which a master source tree is kept on a server host, and client hosts run `coda` to get themselves in sync with the server. The approach taken requires significant system administrator action to set up, such as modifications to files like `/etc/services` and `/etc/inetd.conf` on a Unix machine, and does not authenticate the retrieved files. Our inspection of their `readme` file also leads us to believe that `coda` supports fewer architectures than we do. It also requires a user to log in to the server before any files are sent. Our approach requires no system administrative actions on a client machine.

7 Future plans

Our experience so far with the model we have described has been quite positive. A group of users spread across three continents, and with varying qualities of Internet connection, have been able to keep a large body of code in synch with each other. In the worst case, it can take as much as three or four hours to download the whole REDUCE tree from scratch (about 20 Mbytes). However, once this is done, future updates take only a few minutes, since relatively few files are involved.

⁵<http://escher.usr.dsi.unimi.it/hpux/Sysadmin/coda-2.0>

Now that we have achieved our initial goals, we see several ways in which the efficiency and integrity of the process can be improved.

When an individual file in the REDUCE sources is very large a change to it, however small, leads to a large download cost. In the previous standard distribution of REDUCE, which, as we noted earlier, uses source files to represent packages that are a collection of modules this would have arisen quite commonly: individual package files were often as large as 100 kilobytes, and it would be inefficient to update the whole file if only a few lines are affected. A similar problem arises in the current system with pre-built Windows help files and gnu-info documentation for REDUCE, and with any binary executable or image files that need distribution⁶. To support the few large files we will always have in a complete REDUCE tree, we are investigating the `rsync`⁷ program of Trigdell and Mackerras, which is designed to do just what we want; namely to update a file on one machine to be identical to a file on another machine in a low-bandwidth environment. Their approach divides a given file into sections, with a checksum for each section. This fits nicely with our approach. However `rsync` involves running its own server on a publishing site in a way that is not permitted by some network administrators, and so we are not in a position to use it directly. Thus we need to design an extension to `xport` that achieves as much of the same behavior we can given just dumb servers. We expect that if a first attempt to download a large file fails part way through, but the incompletely fetched part is left installed on a slave site, `rsync`-like behaviour will allow the fetch to continue from close to where it had been interrupted.

If or perhaps when we find that all sites who may wish to maintain parts of REDUCE become able to support cgi-bin scripts or some other server-side computation we can both update these plans for coping with large files and also sharply improve on security.

8 Conclusion

When we started on this work we tried a number of experiments using Web services to move files between various sites. We tried transfers at various times of day within the USA, within Europe and trans-Atlantic. The variation in performance seemed very hard to predict, and could range from a comfortable state where multi-megabyte files could travel from California to Germany and back at some times through to occasions where attempts to fetch just a few kilobytes of text stalled and eventually failed with time-outs. We find that on occasions when networks are running smoothly our automated update process can fetch and install a complete REDUCE development tree on a previously empty computer in an hour or so. The fact that if a transfer is interrupted the update can be restarted without any fuss has proved very helpful. Once a system has been installed the amount of material fetched when a remote site asks to be updated will depend very strongly on how long it has been since that site last ran our scripts. For those who refresh their systems very frequently it will tend to amount to a few hundred kilobytes per week (the typical bulk of the weekly `shar` files used previously), but those who wait longer will usually not see a vast increase on this since often it will be the a compact cluster of REDUCE modules that are being worked on at any one time and so only these will change. As compared with the `shar` distribution our new model arranges that temporarily inactive developers save bandwidth by not receiving (and not having to process) multiple incremental versions of rapidly-changing modules.

⁶Of course the main REDUCE executables are at present always re-created locally by the site that fetches files

⁷`ftp://samba.anu.edu.au/pub/rsync`

Our section on future plans shows that the system as we now have it is not yet perfect, but it is already in service and has been behaving fairly well. Our next concerns must be to refine it to a state where it can be used to support the whole general REDUCE community, not just the smaller group of developers.

9 Acknowledgement

Much of the discussion which moved this work from a vague idea into a concrete project happened at the 1996 REDUCE developers' meeting, Karlsruhe, September 1996. The contributions from W. Neun and E. Schrufer were especially helpful. The other REDUCE developers who are now trying our system (and occasionally suffer when it does not do what they need) have also provided very useful feedback.

References

- [1] DMITRIEV, M. A. The CSL Lisp system as a Web language. Dissertation for the Diploma in Computer Science, University of Cambridge, July 1997.
- [2] DONGARRA, J. J., AND GROSSE, E. Distribution of mathematical software via electronic mail. *Comm. ACM* 30 (1987), 403–407.
- [3] GARFINKEL, S. *PGP – Pretty Good Privacy*. O'Reilly & Associates, 1995.
- [4] GRISS, M., BENSON, E., AND MAGUIRE, G. Q. PSL: a portable Lisp system. *Conf. Record of the 1982 ACM Symp. on Lisp and Functional Prog.* (1982), 88–97.
- [5] MARTI, J. B., HEARN, A. C., GRISS, M. L., AND GRISS, C. Standard Lisp report. *Sigplan Notices, ACM* 14 (1979), 46–68.
- [6] MATOANE, M. A Web-browser interface for the display of algebraic formulae. Dissertation for the Diploma in Computer Science, University of Cambridge, July 1997.
- [7] NORMAN, A. C. Compact delivery support for REDUCE. *Journal of Symbolic Computation* 19 (1995), 133–143.
- [8] RIVEST, R. RFC 1321: The MD5 message-digest algorithm. Tech. rep., RSA Data Security, Inc., 1992.