Chaos Out of Order: A Simple, Scalable File Distribution Facility For 'Intentionally Heterogeneous' Networks

Alva L. Couch – Tufts University

ABSTRACT

In large networks, heterogeneity in hardware, operating systems, user needs, and administrative responsibility often forms boundaries that inhibit sharing of information, expertise, and responsibility. These boundaries can divide networks into 'feudal fiefdoms' of administrators, each with a disjoint domain of responsibility. DISTR is an easy-to-use file distribution tool for homogeneous networks that also provides controlled file transfer between disparate architectures and administrative domains. Using DISTR, administrators of unrelated networks can collaborate to reduce duplication of effort while retaining control of their own networks. DISTR's controls allow collaboration, cooperation, and camaraderie to evolve, not from grand and imposed designs, but from informal and serendipitous commonalities of mission and purpose.

Introduction

Computers, like people, have to share a common language in order to communicate. To share information, they have to agree on the meanings of the files they share. Unfortunately computers, like people, are often prohibited from communicating by what are almost 'ethnic' boundaries: diverse hardware, diverse operating systems, or even diverse user and administrator needs.

While we are all familiar with heterogeneous networks of machines with diverse hardware or operating systems, there are also networks with homogeneous hardware and operating systems but diverse configurations or administrative responsibilities. Users often need many different configurations of the same hardware and operating system. An often overlooked form of heterogeneity is that induced by lines of authority, control, and responsibility. Mimicing the structure of a large organization, a large network of relatively identical machines is commonly organized as several 'feudal' subnetworks, or 'fiefdoms,' each with a different primary administrator or administrative group.

Distributing shared information is relatively easy when one does not have to cross boundaries induced by disparities in hardware, software, user needs, or administrative responsibility. There are plenty of solid approaches to distributing databases, files, and software. However, whenever one needs to cross even one of these boundaries, many problems arise. Different operating environments require different formats for the same information, but formatting information correctly is the least of our problems. It is much more difficult to deal with user requirements and administrative boundaries. Current tools for file distribution cannot be configured at the level of detail necessary to describe and respect these boundaries.

In this paper, I describe a new tool for file distribution, DISTR, specifically designed to deal with heterogeneity in hardware, software, user needs, and administrative boundaries. Written in Perl-5 [21] for portability, DISTR has a simple syntax that eases simple file distribution tasks in homogeneous networks. For advanced users, the functions of DISTR can also be extended by writing custom Perl scripts. These scripts can authenticate data transfers, keep records, distribute files to large networks quickly, and transform data appropriately both before and after each transfer through a heterogeneity boundary.

Distribution Systems

There are currently many ways to distribute the contents of files on a network. These include generic tools that will distribute any kind of file, and domain-specific tools that distribute only particular kinds of files or work only within particular kinds of networks.

The simplest kind of file distribution is to use a command that copies a file from one place to another. Remote copy commands such as the UNIX remote copy command `rcp` and a recent improved version `rsync` require the user to have an account on the target machine and the rights to modify the file to be copied. To update system configuration files, the user must have `root` access to the target machine. Secure remote command facilities including SSH [23] better insure the identity of the updater but suffer from the same need to grant `root` access. Alas, this simple approach works for very small networks but is not scalable to large networks or applicable to non-UNIX systems.

General-purpose UNIX file distribution schemes such as RDIST [7] automate the process of using remote commands on a server to copy files to remote clients. RDIST can both update files and execute remote commands necessary before the new versions take effect. Reverse file distribution systems like RevRdist [25] and PC-Rdist [17] perform the same function but reverse the roles of client and server. These execute on each client at boot time or some other convenient time and copy files from a server to the client, accessing the server's files using a file sharing mechanism such as AppleShare or the Network File System(NFS) [20]. Before updating a client file, all of these tools check that it needs updating by comparing its size, creation time, and other attributes with those of the master copy on the server. The goal of these tools is to allow one administrator to exclusively control more machines than is usually possible.

Software package distribution utilities such as Depot [6, 15], opt_depot [1], Depot-Lite [19], Cicero [4], and their variants accomplish the same task as RDIST, but limit their problem domains to the special problem of distributing usable software. This limitation allows these tools to perform more complex tasks than generic tools like RDIST while requiring less effort to configure. Variant file distribution schemes such as Local Disk Depot [22] add the ability to customize each client machine's software environment, to avoid installing unused software and conserve disk space.

Configuration trackers [3] document the contents of specific UNIX configuration files on clients in a heterogeneous UNIX environment. A database on a server describes the similarities and differences between clients. A configuration tool uses remote commands to assure that clients' configurations agree with the database. This again is what RDIST does, limited to a very specific problem domain to simplify usage and avoid distribution errors.

Domain-specific UNIX database services such as NIS [20] and NIS+ [18] automatically provide data from tabular databases on a network. NIS client machines must choose one server to provide database information. NIS+ adds the ability to query multiple servers and administrate large databases in distributed, manageable pieces. As both NIS and NIS+ only work on databases in which each line of a table has a unique key, neither is suitable for providing files without a tabular structure.

Portable domain-specific database services such as the Lightweight Directory Access Protocol(LDAP) [24] and Domain Name Service(DNS) [2] overcome many of the portability limits of NIS and NIS+. These are designed to provide information to all hosts in a heterogeneous environment but are too specialized and optimized for their problem domains (mail addresses and internet names) to be useful for general-purpose distribution of any other information.

A pattern develops in examining all these approaches. All except NIS+ take their information from one master server, even if slave servers replicate its contents. Each approach is either good for distributing a specific kind of information to different kinds of client machines, or all kinds of information to roughly the same kinds of client machines. None of these tools addresses the general problem of distributing the contents of arbitrary files in heterogeneous domains containing a mix of UNIX and other operating systems. There is a good reason for this; the general file distribution problem is very difficult!

Developing trust

"It would be nice" if we could all trust each other and cooperate on what is obviously a common goal. Unfortunately, trust is difficult to encourage among diverse groups. It is especially difficult to encourage when trusting another group to provide services means giving all of them the ability to change anything at all on any machine in your own network! But this is precisely what most current file distribution tools require. Trust is much easier to develop incrementally from small, serendipitous commonalities of goals and purposes. With proper tools, trust can grow from 'grass roots' upward, from good experiences in allowing cooperative people the access they need to make a difference.

For example, many fiefdoms are small enough that they do not run their own name service. This is instead handled by a central service group that determines, e.g., the contents of the name service configuration file `resolv.conf` for the feudal network. But when this file changes, the central group may not have privileges to install it within the fiefdom. Typically someone from the central group mails each new version to someone inside each fiefdom and tells the internal person to install it.

This situation wastes the time of both the central group and the fiefdom administrator. Central services should be able to change `resolv.conf` without changing anything else. The fiefdom administrators should be able to trust central services to update this file, regardless of whatever else they think about central services.

Desires

After a several year struggle using existing file distribution tools and writing front-end tools that extend their capabilities, I set out to implement a new file distribution tool DISTR that does not suffer from their limits.

My priorities were quite simple at the outset. I wanted a file distribution mechanism that allows client machines complete control over which files they accept. I have had many problems with particular users who need customizations outside the limits of my configuration management scheme. In a hurry, I wanted something that will allow me to turn distribution actions on and off at the client end and modify them according to personal taste.

I wanted a file distribution mechanism that is symmetric in the sense of supporting both master to slave and slave to master requests. I like the master to slave administration model, but I have historically had much trouble keeping all hosts synchronized when lone hosts suffer hardware failures and miss receiving files.

I also wanted a mechanism that utilizes portable file names rather than names particular to one operating system. I chose the dotted name notation of many tools as a starting point. Each portable name is a point of sharing between two hosts; regardless of whatever else they can agree upon, they can agree upon a name for the file being exchanged.

Finally, I wanted a tool that is very easy to use at the outset and hides advanced features from novices. I did not want to have to think very much about routine distribution tasks. But I also wanted to be able to perform arbitrarily complex tasks using the tool with more effort.

Beginning from these needs, I crafted DISTR. What evolved is a much more complex entity than these needs prescribed.

Using DISTR

DISTR's basic usage is simple if a bit cumbersome. In order to use DISTR, one has to install DISTR's daemon `distrd` and DISTR's user front end `distr` on each host to be involved in providing or receiving files. The daemon runs at all times awaiting requests. A host that provides a file is called a *server* while a host that receives one is called a *client*. In DISTR, there is no real distinction; any host can be a client of some files, a server of others, and a server and client of still others.

Next, one creates a configuration file `distr.conf` on each host describing the files that this host is allowed to receive or provide. On each server, `distr.conf` contains lines like:

```
aliases.export.source =
        '/usr/lib/aliases';
aliases.export.clients =
        ['mine', 'yours'];
```

These particular lines give DISTR permission to export the local file `/usr/lib/aliases` to two clients `mine` and `yours`.

The configuration file of each client receiving this file must have a matching set of lines like:

```
aliases.import.target =
        '/etc/mail/aliases';
aliases.import.servers =
        ['theirs'];
```

These lines give the client permission to accept a file from the server `theirs` and put it into the local file `/etc/mail/aliases`.

With configuration files in place, the user then requests distribution actions by executing commands like

```
# distr aliases.import
```

on a client to import the alias file from a server or

```
# distr aliases.export
```

on a server to export the alias file to all clients.

Using DISTR is very different from using RDIST. Both client and server must have configuration files, and these must agree on what gets exported from one machine and imported into another. For a file transfer to be able to occur, server and client have to agree on a *portable name* (such as `aliases`) both will use to refer to the file, and each must define enough *parameters* for the transfer to allow it to occur. Parameters have names that are prefixed with the portable name of the file.

At bare minimum, server and client have to specify two parameters. On a server, `export.source` is the name of a file to provide and `export.clients` is the familiar RDIST-like list of all clients to which to provide it. On a client, `export.target` is the name of a file to accept and `import.servers` is a list of all servers from which the client can potentially obtain the file. One denies a host the privilege to install a file by simply omitting the host from the list of authorized servers for a file or, even more simply, omitting the file name from all DISTR declarations for the client.

Once this information is specified, one can initiate a file transfer from either host. Like RDIST, one can tell the server to transfer a file to clients. Like RevRDist, one can tell a client to fetch a file from a server. Unlike RevRDist, DISTR will search for a valid copy of the file sequentially on all servers on its list, stopping when it finds and manages to fetch one.

Simplifying Syntax

DISTR provides several syntactic features to make configuration file syntax less cumbersome. Complex specifications are easier to type by using a name scoping notation. The above server configuration can also be written as

```
aliases.export {
 source = '/usr/lib/aliases';
 clients = ['mine', 'yours'];
}
```

Inside the braces, names are prepended with the prefix given before the braces. Scopes nest to arbitrary depth, so one could also write the server configuration as

```
aliases { export {
 source = '/usr/lib/aliases';
 clients = ['mine', 'yours'];
}}
```

Using Inheritance

DISTR's parameter values may be defined or inherited. A defined parameter has a specific value listed in DISTR's configuration file. An inherited parameter gets its value from another defined parameter. A name with no defined value inherits the value of its *longest defined suffix*. If `c.d` is defined and `b.c.d` and `a.b.c.d` are not, then `a.b.c.d` inherits the value of `c.d`. This allows the user to assign default values to parameters rather than typing explicit values for all distribution actions.

For example, we can write:

```
export.clients =
    ['mine', 'yours'];
aliases.export.source =
    '/usr/lib/aliases';
group.export.source =
    '/etc/group';
```

to implicitly export *all* source files to both clients. Attributes `aliases.export.clients`, `group.export.clients`, and every other attribute whose name ends in `export.clients` inherit their values from the 'global' definition for `export.clients`. Thus these attributes do not need to be specified explicitly.

In turn, the rest of the configuration file need not mention `export.clients` again and can consist only of mappings between portable names and names of files to export. In homogeneous environments with one server and clients that all receive the same files, DISTR configuration files are thus much shorter in length than comparable configuration files for RDIST.

This notion of inheritance is almost exactly *backward* from notions of inheritance in object-oriented languages such as Java [11] and JavaScript [12] that use the same kinds of namespaces. Object-oriented inheritance helps one specify rules that define classes of objects but does not create any instances of those classes. Our inheritance functions instead help us specify *instances* of classes that never get explicitly defined as classes! Rather than defining a set of classes each of which can have an unlimited number of instances, DISTR's syntax defines an unlimited number of instances, leaving the classes implicit in that definition!

DISTR's Syntax

DISTR's configuration file defines values for selected attributes. Each attribute is represented by a dotted name and can represent either an action or parameter. The value of an attribute is the value of a Perl expression. This value can be any kind of Perl scalar, including a reference to an array, to an associative array, or to a function. An attribute whose value is a reference to a Perl function is called an *action*, while an attribute with a non-function value is called a *parameter*. Internally the only distinction between actions and parameters is that when other actions request their values, actions are implicitly invoked as functions while parameter values are simply returned to the requester. The set of all defined attribute names and their values is called DISTR's *namespace*.

DISTR distributes files solely by invoking actions defined in its namespace. These actions query the namespace to determine their own operating parameters. As inheritance works for actions as well as parameters, an action can be invoked by many different names with differing prefixes. The prefix on the name by which an action is invoked determines the parameters it fetches from the namespace.

DISTR provides two default actions `export` and `import`. These are never invoked directly, but always through inheritance. The names by which they are invoked determine which files are exported or imported. For example, invoking `export` as `aliases.export` (through inheritance) causes it to use the parameters `aliases.export.source` and `aliases.export.clients`. The values of these may themselves be inherited. If `export` is called by any other name its parameter names change to match.

Customizing DISTR

In configuring DISTR, the user can override the definitions of *any* action, including default ones such as `export` and `import`. However, these are very complex actions. For proper function of DISTR, these actions have to have roughly the same form regardless of customizations. For this reason, high-level operations such as `import` and `export` are phrased in terms of lower-level component operations. These lower-level operations can be individually overridden to alter specific parts of DISTR's behavior. The `import` and `export` routines are actually *algorithmic skeletons* [5] that hide the complexities of DISTR's functions while allowing one to customize anything easily.

For example, the default `import` action looks like this:

```
import = sub {
 if (&some('import.needed')) {
  if (&some('import.authentic')) {
   if (&some('import.before')) {
    if (&some('import.method')) {
     &some('import.afterSuccess');
    } else {
     &some('import.afterFailure');
    }
   } else {
    &some('import.afterBeforeFailure');
   }
  } else {
   &some('import.afterDenial');
  }
 }
};
```

The DISTR library function `some` looks for a parameter value relative to the name under which the current action was invoked. If `some` finds a value representing an action, it executes that action and returns the result of execution. If `some` finds a non-action value, it simply returns that value. The result is that the above code executes several actions in order:

- `import.needed` determines whether an import is needed at this time.
- `import.authentic` determines whether importing should be allowed.
- `import.before` prepares for importing, including backing up old versions, etc.
- `import.method` actually imports the file.
- `import.afterSuccess` does import post-processing if the import succeeds, such as updating other related databases.
- `import.afterFailure` performs cleanup after import errors, including restoring old versions of files.
- `import.afterBeforeFailure` performs cleanup after fatal pre-import errors.
- `import.afterDenial` takes action concerning security failures, including notifying administrators, etc.

By default, only `import.needed`, `import.authentic`, and `import.method` are defined as actions. The rest have value 1 (to disable them) and are provided explicitly for the purpose of customizing the import process.

The typical administrator, concerned with only a few of these phases, can choose to override actions in chosen phases and retain defaults for every other phase of importing. For example, to run `newaliases` after importing a mail aliases file, one would write

```
sendmail.aliases.import.afterSuccess =
  sub {
   system('/usr/ucb/newaliases');
  };
```

or its equivalent for your own flavor of UNIX. One of the nicest things about DISTR's configuration model is that the administrator of a local machine who does not control alias file distribution can add local modifications before compiling the file, e.g., concatenating the alias file with a local one before making it take effect.

PGP Authentication

DISTR's default authentication is host based like RDIST's. Clients maintain lists of hostnames and addresses from which requests will be accepted, though these lists are private to DISTR and not used for other purposes. DISTR also allows authentication of the creator of a distributed file as a person, using PGP signatures to determine the identity of the creator.

To authenticate every incoming file via PGP 2.6.2, one can write

```
import.authentic = \PGPauthentic;
```

to use the builtin PGP authentication function for every import transaction. One must also define two PGP parameters: list `signers` of PGP names of people allowed to provide files for distribution and the name of a keyring file DISTR can use to validate file signatures. These parameters are inherited and can change for different imported files.

Files to be authenticated must be PGP signed on the source machine using detached signatures. A file's detached signature, if available, is forwarded as part of any export request or import response. If PGP authentication is in effect, a file will only be imported if its signer matches one of a list of PGP users authorized to import files.

DISTR uses PGP for authentication only. Files are not encoded or encrypted using PGP, but remain in plaintext. There are two good reasons for this apparent oversight. Detached signatures are easy to use and do not prohibit a host receiving them from ignoring them and falling back to host-based authentication if the host does not have PGP. Encrypting each file using PGP would also require that the receiving daemon have access to a *private* key for purposes of decrypting it. Since there will not be a user available to type a passphrase, that private key would have to be stored on the host somewhere and vulnerable to discovery. In my view this use of PGP encryption provides more of an illusion of security than actual security. DISTR transmits all its files in plaintext and is not suitable for transferring sensitive information, and no use of PGP will solve this problem satisfactorily.

While this approach solves the problem of authenticity, there is still a serious security problem in using this very simple form of PGP authentication. Any file, once signed, can be successfully included in any request the signer is permitted to make. This means that a devious person with possession of a signed file can use replay attacks to corrupt every file the signer is permitted to change!

There is only one stateless PGP solution to this problem that is under development. The signer can also provide a 'certificate of intent' listing where the file should be installed. If this is also PGP signed, a host receiving the file and certificate can insure that it is utilizing the file in the way the creator intended.

Even this is insecure, because a devious person with access to an older version of a file, its certificate, and their signatures can use them to initiate a rollback attack that resets a target file to an old configuration. This can re-open security holes or deny services based on improper machine configurations.

There is again only one (almost) stateless PGP solution to this problem that is under development. DISTR can ask the initiator of a request to sign a file describing the request itself. If this file contains a time stamp and time limit on the request, and if receiving hosts check that the request has been sent between those limits, rollback attacks can be prevented.

There are too many 'if's in this discussion. My conclusion from this is that PGP provides just a bit better security than the default host-based security, and in general is not particularly suitable for providing security in this context. Of course, with DISTR's modular structure any new available form of security can be implemented relatively quickly. One should be!

Remote execution

DISTR also can provide general-purpose remote execution capabilities. As `distrd` runs as `root`, any actions specified are privileged unless otherwise noted. This means that I can easily configure DISTR to import and execute a Perl script in a somewhat secure manner, by writing:

```
penguin.import {
 target = "/tmp/penguin$$";
 needed = 1;
 signers =
   ['Alva Couch <couch@tufts.edu>'];
 authentic = \PGPauthentic;
 afterSuccess = sub {
  my $name = &some('import.target');
  system("/usr/bin/perl $name");
  unlink $name;
 };
 afterFailure = sub {
  my $name = &some('import.target');
  unlink $name;
 };
}
```

The result of this simple hack is to force importing of any file with the portable name `penguin`, authenticate it as mine, and if authentic, execute it as a Perl script. This is not as secure as the real PENGUIN [13] remote execution utility for Perl scripts, which executes its scripts in a controlled, limited environment based upon privilege specifications. DISTR's scripts are executed with no limitations.

I do not recommend doing this. As above, this mechanism is subject to replay attacks. Once a file is signed, anyone who can gain ownership or capture it on the network can forward it to the daemon for execution.

Handling Heterogeneity

Effective communication between servers of diverse hardware and software architectures requires careful configuration of DISTR. *Any attempt* to move information from one architecture to another requires that one:

1. Understand the forms that information will take on either side of the architectural boundary, including files and their formats.
2. Design a 'portable format' for the information that can be transformed into the forms needed on either side. This may consist of information from several different files on each side of the boundary, appropriately combined.
3. Develop filters that create the portable format from files on the server side, and that transform the portable format into desired files on the client side. These filters form the bulk of DISTR's `export.before` and `import.afterSuccess` methods.

If one is lucky, the native form of information on one side of the boundary can be used as the portable form on the other. For example, suppose we are sending a UNIX database to an NT workstation. If there are many NT stations and few UNIX servers, it makes sense for the database to get translated once on the UNIX server, then propagated to all the NT stations in native form.

Scaling DISTR

DISTR's distribution algorithm can also be adapted to update large networks in minimal time by a relatively simple modification. If servers are linked by DISTR, and each server has both `import` and and `export` specifications for each file being distributed, one can configure each server to export whatever it imports by writing

```
import.afterSuccess = sub {
  some('export.initiate');
};
```

The `export.initiate` method causes the server to invoke the appropriate `import` on other servers previously listed as clients of this one. Then, if configurations of servers are arranged so that one server updates all others, each server will independently update all its clients.

One must of course take care to insure that this distribution process does not create an infinite loop, either by design of the distribution topology or by insuring that loops otherwise terminate. E.g., one can prevent loops by insuring that `import.needed` aborts the exporting process when trying to update a file that is already up to date, as in RDIST. As with any recursive propagation technique, failure to insure this carefully can result in a network storm and network overload.

A safer scalable approach, though somewhat strange in semantics, is to write:

```
export.before = sub {
    some('import.initiate');
};
```

This causes servers exporting a file to synchronize with *their* servers before exporting it, so that the file gets propagated from some master server all the way down to the client. However, this can also cause version skews on the network. A request from a client causes a chained update of a *path* of servers between it and the master server, leaving other servers away from the path alone, even if they require updating. This causes unpredictable version skews in server configurations if the servers use the file in their running configurations. The technique is entirely safe, however, if version skews are acceptable at leaf nodes, and if servers do not utilize distributed files in their own configurations. This is the case, e.g., in microcomputer networks where users are responsible for initiating their own updates.

Scaling Vulnerabilities

The security issues discussed above are even more important when DISTR is configured to broadcast files to a network. RDIST-like host-based authentication is vulnerable to spoofing attacks. DISTR's PGP authentication for imported files provides little help due to its vulnerability to replay attacks. A devious person can use these techniques to compromise a *network* configured for recursive propagation of files.

DISTR does not currently solve this problem. The two signed certificates of intent and time limits proposed above would only partially solve it. If a user makes a mistake in a file, signs it, distributes it, discovers the mistake, and redistributes it before the time limit on the original certificate, a devious person can set up a distribution chain for the incorrect file *in competition* with the corrected version.

What is Scalability?

Scalability typically refers to the way the performance of a software tool or algorithm changes as the number of computers it manages or utilizes increases without bound. While this definition is well suited to the needs of people analyzing network hardware or parallel computing algorithms, it is not so well suited to analyzing system administration tools. In algorithm analysis, the emphasis is on *performance* and *size*, while in system administration our emphasis is on *usability* and *complexity*. A usable tool must of course perform reasonably, and a large network is indeed a complex one. But system administrators typically have little interest in fast, unusable tools and may manage relatively small but relatively complex networks (such as my own).

For a system administrator, therefore, a scalable tool is one that handles problems with varying complexities and remains usable and cost effective at all scales of task complexity, whether or not complexity is caused by numbers of machines or environmental heterogeneity. A tool's scalability thus refers not only to its asymptotic performance on large networks, but also to its ease of use in performing small tasks, and the ease with which one can learn to perform small tasks with it.

My best example of a scalable configuration language is SLINK [8, 9], intended for managing images of local file repositories. Simple filesystem manipulations require only simple commands, while more complicated actions require more involved commands and a complete understanding of SLINK's protection model. In SLINK's case, the command syntax is crafted to discourage undesirable actions by making them more complex to specify [10]. In DISTR's case, instead, syntax is crafted so that homogeneous, domain-specific actions are easier to specify than heterogeneous, domain-bridging ones.

DISTR's language is designed to be the absolute minimum one needs for dealing with heterogeneous environments. Basic functions, including authentication, pre-processing, post-processing, and transport can be configured by specifying skeleton functions with a minimum of effort. Thus DISTR's *configuration language* is scalable in the sense of the above definition; easy to use for simple tasks and capable of performing any task with effort.

While DISTR's distribution strategy and language are scalable, its security mechanisms based on host address and PGP are not. Solving this problem will require mechanisms other than PGP.

How DISTR Works

To accomplish distribution actions, a DISTR daemon `distrd` runs on all hosts managed by DISTR. This daemon interprets a local configuration file `distr.conf`, reads requests off the network, and responds appropriately to each. Requests are made either by other daemons or by DISTR's front end user program `distr`.

Requests to the daemon contain not only the full name of the action requested (`import` on the remote host for exports, `export` for imports) but also everything known about the action on the requesting host, including all parameters specified for the action in the requesting host's configuration file and the contents of local files if appropriate. To transmit this data, DISTR uses a custom Perl library `Data::Pipeable` that can transmit any acyclic Perl reference structure through a pipe and reassemble a duplicate on the other end of the pipe. This library also allows DISTR to embed the contents of files of arbitrary length into the argument list sent to the remote machine.

`Data::Pipeable` is based upon the idea of the Comprehensive Perl Archive Network (CPAN) [16] library `Data::Storable`, which allows storage and retrieval of Perl data structures to and from disk files. Due to an improved algorithm, the new implementation does not utilize any C code to improve portability. Due to a lack of any sensible way to reconstruct them, as well as the potential security problems involved in trying, references to functions are not transmitted.

Both import and export requests are simple, stateless transactions consisting of a single request and a single response. Currently each `export` request contains the file's portable name and the target file's size, protection, owner, mode, and modification time. If these attributes do not match those on the server, the file is returned along with its own attributes and PGP signature if available. The returned file is then checked locally for authenticity and installed if authentic. Each `import` request contains the file to be installed and all information known about it. This information includes the file's size, protection, owner, mode, modification time, and PGP signature if that is available. The remote client checks whether the file should be installed and installs it if installation is needed and allowed.

I have come to regret this design. I will soon be changing the `import` protocol to a two phase protocol (like RDIST's) with an initial probe for validity and a second phase in which the file is sent. This will not only save network bandwidth but also make the daemon more resistant to denial-of-service attacks where someone sends large unauthenticated requests to the daemon to keep it from processing valid requests. Currently invalid requests can fill filesystems with unauthenticated files.

DISTR's front-end program `distr` actually has a much more complex task than the daemon. Since the structure of inheritance defines an unlimited number of actions, the front end's job is to decide which actions to invoke. It does this in a very simple way, by looking at their attributes. An import event is meaningless without designation of a target file, while an export event is meaningless without a source file. When asked to import a file or files, the front end matches its request against all target files it knows about and imports only those targets. Likewise exports are performed only for defined source files.

Limitations

While DISTR may seem an advanced tool, it has many limitations imposed both by a need for simplicity and a lack of implementation time.

DISTR's daemon is inherently serial. It processes one request at a time, then looks for the next. This is both a limitation and a feature. While there is no compelling reason why the daemon cannot fork, I do not want it forking in my network! File distribution should never inhibit day to day operation of a workstation. Even if I do provide the daemon eventually with the ability to fork, I will limit it to at most four concurrent operations. This change, however, will change the semantics of DISTR. Currently, one can invoke serial updates on one host that are guaranteed to happen serially on all targets. This will not be so if the daemon can fork, and configuration files will have to be modified for this possibility.

The PGP implementation used in DISTR is version 2.6.2 using a variant of the front end written for PGP by Felix Gallo in implementing PENGUIN. This is a very limited implementation and provides the bare minimum of functionality. This is simply what was available at the time DISTR was written. I expect to replace this with a better PGP interface when possible.

DISTR's components, both client and daemon, scan the local configuration file as a text file to initialize themselves. Since this file contains Perl code, both of these compile that code on the fly. A lack of finesse in compiling each attribute's value results in quite cryptic error reporting, a deficiency I plan to remedy shortly. While DISTR in principle will execute on any host supporting Perl and daemons, it has only been tested for UNIX. Further development is needed to make it truly portable, including using improved PGP library functions.

While DISTR provides a transport layer suitable for communication between diverse hosts, and a portable namespace that hosts share, DISTR specifies nothing about exactly what names hosts will agree upon and what transformations will occur in translating information from one format to another. Semantics-preserving transformations, such as distributing UNIX groups for use in NT, must be hand-configured by the user.

Security

Despite my best efforts, DISTR is frightfully insecure. Unfortunately, DISTR's serious security problems are the same as would be encountered when implementing any scalable stateless security mechanism on a network. It is quite easy to compromise it with a replay attack.

For DISTR to be scalable, we cannot include mechanisms that compromise that scalability. This means that we cannot defeat replay attacks the easy way by assigning serial numbers to requests and enforcing simply increasing serial numbers. Each host would have to remember the last request serial number from each other host, and rebuilding a host after a crash (when DISTR is most useful) would be problematic.

Call me irresponsible, but DISTR's default running configuration is very insecure. The default authentication is host-based and RDIST-like, with all the security problems that implies, including being prone to address spoofing attacks. The reason for this is that the average user looking for something like RDIST will be more likely to utilize DISTR if it acts something like RDIST in the beginning, and less likely to utilize it if one has to learn everything about PGP first.

This has serious implications. If a naive user fails to heed my warnings and implements recursive propagation on a large network without PGP-authenticating the files at least, that user's whole *network* will be prone to spoofing. If that same user implements the PENGUIN lookalike code given above without using PGP, all bets are off on the security of the network. It is *very* easy to completely compromise the security of DISTR in the configuration file. Since DISTR never *enforces* security limits, but makes them optional, it can never be considered completely secure.

DISTR's daemon is also prone to denial of service attacks based on message volume. Since the current protocol sends files to be distributed as part of the request, the daemon stores them in a temporary location before determining what to do with them. If a devious person sends very large files the daemon will happily fill up `/tmp`. Though it is more difficult to get DISTR to install those files anywhere crucial, this problem is very annoying and will be fixed by revising DISTR's protocol.

Critique

It is difficult to evaluate DISTR against other tools providing the same functions because DISTR's priorities are so different. Rather than configuring just a server, the user of DISTR must configure both servers and clients. This requires construction of one configuration file per *type* of client, as well as bootstrapping each client with the daemon and its configuration before using DISTR. Whether this is worth the effort depends upon one's goals. If the extra functionality of DISTR is appealing, the extra work, including the bootstrap, is probably worth the effort. If one desires RDIST functionality, then DISTR is much more work to deploy. If one desires remote privileged execution of selected scripts, DISTR's configuration provides the safest way I know to do it.

It is a dubious practice to provide a mechanism for use of standard naming without providing guidance about the naming standard, but this is all my time allows. To avoid massive confusion, users of DISTR have to remain consistent with their own naming standards. The best guidance I can give is to adopt a standard that works like the language, from general to specific, and clearly identifies platform dependencies by levels of the naming scheme. Everything under the name `unix` should refer to transactions specific to UNIX, while everything under `nt` should refer to transactions specific to Windows-NT. For cross-platform transfers, the name of the action should evoke the subsystem being updated, e.g., `groups` for a cross-platform transfer of user groups between different operating systems, `mail` for generic mail information, etc.

It is also a dubious practice to provide a modality for bridging domains without providing any utility functions to help. But I am ignorant of other users' needs. The namespace, however, provides an easy way to categorize and label server and client protocols for accomplishing various kinds of transfers. If DISTR or similar techniques prove popular, I hope users will help one another out in building these translations.

It is also questionable whether a distribution mechanism that requires hand-configuration of its distribution topology is scalable. In a large network, it is a lot of work to tell all the servers about all their clients and vice versa. But I know of no reliable automatic method for determining this information from lists of equivalence classes of clients. The general problem of mapping a distribution scheme optimally onto a given topology is intractable, and it is almost as difficult to precisely describe a network topology devoid of distribution topology as it is to describe the distribution topology itself. Worse, typically we would rather not allow a program to determine which machines are servers and which are clients; we decide which machines are servers beforehand. An alternative to explicitly hand coding the topology seems quite difficult to implement and of dubious value.

Conclusions

We should remember that in our capacities as system administrators we are not networking computers, but people. The first step in this task is to network the people who maintain the computers. But commonly, we ourselves do not manage to bind together into a group that is stronger than the sum of its parts. Lack of clarity in our roles combines with ambiguity in our powers to create situations that pit us against each other instead of against real problems with the network.

DISTR provides a beginning of a new camaraderie between administrators in what used to be opposing factions. There is no need to trust anyone with one's life – one can trust people to do what they do best and have no doubt that this is all they do. We no longer have to give an untrusted person the root password and wonder exactly how bad things can become. And we no longer have the potential to blame people for problems who could not possibly have caused them due to lack of privilege.

One of the most important parts of being a member of a family is learning to protect one's boundaries without excluding others. Through no fault of their own, fiefdoms play the role of outcast family members in a large family. Fiefdoms exist partially because they do not have the tools to protect their boundaries without excluding others. Tools like DISTR are the first step in transforming fiefdoms into families. Living in a good family is a lot more pleasant.

Availability

DISTR 2.0 is in testing and will be available by conference time in the directory ftp://ftp.eecs.tufts.edu/pub/distr. DISTR 1.0, although known by the same name, is quite different in function. It is a front-end to RDIST that implements only archiving and rollback features.

Acknowledgements

David Krumme endured multiple excruciating discussions of functionality during the development of DISTR's predecessors, and gave much valuable feedback and guidance. Scott Corzine gave me extensive advice and design help, and helped me develop the proper paranoic attitude about security and authentication. My spouse Elizabeth endured proofreading of many paper revisions. Special thanks to Remy Evard for last minute help with clarity and references. The name DISTR is not just a pun on RDIST. DISTR is also the name of a primitive operator in the Berkeley implementation of Backus' functional programming language FP. FP's DISTR is also called the 'right distribution primitive'.

Author Biography

Alva L. Couch was born in Winston-Salem, North Carolina where he attended the North Carolina School of the Arts as a high school major in bassoon and contrabassoon performance. He received an S.B. in Architecture from M.I.T. in 1978, after which he worked for four years as a systems analyst and administrator at Harvard Medical School. Returning to school, he received an M.S. in Mathematics from Tufts in 1987, and a Ph.D. in Mathematics from Tufts in 1988. He became a member of the faculty of Tufts Department of Computer Science in the fall of 1988, and is currently an Associate Professor of Electrical Engineering and Computer Science at Tufts. In 1996 he received the Leibner Award for excellence in teaching and advising from Tufts. He has assisted in maintaining the Tufts computer systems for Computer Science teaching and research since 1985, when he was a Ph.D. student, and is currently responsible for maintaining the largest independent departmental computer network at Tufts. He can be reached by surface mail at the Department of Electrical Engineering and Computer Science, 161 College Avenue, Tufts University, Medford, MA 02155. He can be reached via electronic mail as couch@eecs.tufts.edu. His work phone is (781)627-3674.

References

[1] Jonathan Abbey, "opt_depot web site," http://www.arlut.utexas.edu/csd/opt_depot/opt_depot.html.

[2] Paul Albitz and Cricket Liu, *DNS and BIND, 2nd Edition*, O'Reilly and Assoc., 1996.

[3] Paul Anderson, "Towards a High-Level Machine Configuration System," *Proc. LISA-VIII*, 1994.

[4] David Bianco, Travis Priest, and David Cordner, "Cicero: a Package Installation System for an Integrated Computing Environment," http://ice-www.larc.nasa.gov/ICE/doc/Cicero/cicero.html.

[5] Murray Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, MIT Press, 1989.

[6] Wallace Colyer and Walter Wong, "Depot: a Tool for Managing Software Environments," *Proc. LISA-VI*, 1992.

[7] Michael Cooper, "Overhauling Rdist for the '90's," *Proc. LISA-VI*, 1992.

[8] Alva Couch and Greg Owen, "Managing Large Software Repositories with SLINK," *Proc. SANS-95*, 1995.

[9] Alva Couch, *SLINK Manual*, 1996. http://www.eecs.tufts.edu/couch/slink.html.

[10] Alva Couch, "SLINK: Simple, Effective Filesystem Maintenance Abstractions for Community-Based Administration," *Proc. LISA-X*, 1996.

[11] David Flanagan, *Java in a Nutshell*, 2nd edition, O'Reilly and Assoc., 1997.

[12] David Flanagan, *JavaScript: the Definitive Guide, 2nd edition*, O'Reilly and Assoc., 1997.

[13] Felix Gallo, *Penguin-3.00*, available from CPAN [16].

[14] James Murray, *Windows-NT SNMP: Simple Network Management Protocol*, O'Reilly and Assoc, 1997.

[15] Kenneth Manheimer, Barry Warsaw, Stephen Clark, and Walter Rowe, "The Depot: a Framework for Sharing Software Installation Across Organizational and UNIX platform boundaries," *Proc. LISA-IV*, 1990.

[16] Jon Orwant, "Welcome to the Comprehensive Perl Archive Network!" http://www.perl.com/CPAN-local/CPAN.html, 1997.

[17] Pyzzo Software, Inc., "PC-Rdist Software Distribution System," http://www.pyzzo.com/pcrdist/.

[18] Rick Ramsey, *All About Administering NIS+*, Sun Microsystems Press.

[19] John P. Rouillard and Richard B. Martin, "Depot-Lite: A Mechanism for Managing Software" *Proc. LISA-VIII*, 1994.

[20] Hal Stern, *Managing NFS and NIS*, O'Reilly and Assoc., 1991.

[21] Larry Wall, Tom Christiansen, and Randall Schwartz, *Programming Perl, 2nd edition*, O'Reilly and Assoc., 1996.

[22] Walter C. Wong, "Local Disk Depot – Customizing the Software Environment" *Proc. LISA-VII*, 1993.

[23] Tatu Ylönen, "SSH (Secure Shell) Remote Login Program," http://www.cs.hut.fi/ssh/.

[24] "Lightweight Directory Access Protocol," http://www.umich.edu/ rsug/ldap/.

[25] "RevRdist Home Page from Purdue U," http://www.purdue.edu/revrdist/.