7

The Siphon: Managing Distant Replicated Repositories

Francis J. Prusker
Edward P. Wobber

May 91

Publication Notes

This report is a revised and extended version of the paper entitled *The Siphon: Managing Distant Replicated Repositories*, by the same authors, published in the Proceedings of the IEEE Workshop on Management of Replicated Data (Nov. 1990).

Edward P. Wobber is with the Digital Systems Research Center, Palo Alto, California, USA.

Abstract

The Siphon is intended to facilitate joint software development between groups working at distant sites connected by low bandwidth communication lines. It gives users the image of a single repository of individually manageable units, typically software or documentation components. Users can lock and modify each unit, the result being propagated automatically to all sites. The repository is replicated at each site, and possibly on multiple file servers for greater availability and reliability. A Siphon has been in operational use since January 1989 between three Digital research laboratories. We presently share a 2.5 GB repository of source code, libraries, documents, and executable files.

Résumé

Le Siphon facilite la coopération des équipes qui développent du logiciel sur des sites distants et connectés par des lignes de communication à faible débit. Le Siphon donne aux utilisateurs l'image d'une bibliothèque unique de composants logiciels. Chaque composant peut être individuellement verrouillé et modifié, les modifications se propageant automatiquement aux autres sites. La bibliothèque est dupliquée sur chaque site, avec éventuellement plusieurs serveurs de fichiers par site pour améliorer la disponibilité et la fiabilité. Un Siphon est opérationnel depuis Janvier 1989 entre trois laboratoires de recherche Digital. Ces laboratoires partagent actuellement une bibliothèque de 2,5 milliards d'octets, composée de programmes source, de librairies, de documents et de fichiers exécutables.

Keywords

Replicated data, software repository, wide area network, distributed software development, file servers.

Acknowledgements

We are grateful to Mark Manasse, who first suggested the idea of on-line package sharing with PRL; to Andrew Birrell, for his keen distributed systems insight; to Kathleen Milsted and Greg Nelson for their comments on this paper; and to Patrick Baudelaire and Henri Gouraud who pushed the idea of an automatic update tool between SRC and PRL.

Contents

1 Background

Soon after its foundation, Digital's Paris Research Laboratory (PRL) decided to use the Topaz [3] software environment developed at the Systems Research Center (SRC) in Palo Alto. This decision presented both labs with a unique opportunity to discover new techniques to support software development over a wide-area network. A joint project was formed whose goal was to present the image of a single shared software repository to researchers at both sites, with a minimum of interactive delay due to transatlantic communications. This paper describes the system, called the *Siphon*, that emerged from this effort.

2 User Model

The image evoked by the word *Siphon* is quite intentional. The best way to view the system is to imagine a set of replicas for a shared data repository which are fully connected, but by narrow data paths. Updates can be made to any replica, but the result ultimately propagates to the others at the maximum rate allowed by the connecting pathways.

The shared repository is divided into individually manageable units. These units define the granularity of all write operations. For each there exists a mutual exclusion lock controlling the creation of new content. A writer must first obtain the lock, then update the local replica, and finally release the lock. A background daemon then propagates the update to remote replicas. All updates are atomic: partial state is never visible, even when an update is in progress.

Reads from the repository are always satisfied at the local replica. Since operations involving locks potentially require high latency network access, it is important that most reads be feasible without locking. This seems perfectly adequate for applications like browsing software source and obtaining load images. For these applications, repository reads amount to unmoderated file system calls which don't involve the Siphon.

When new content is written, it appears at the local replica immediately. However, distant replicas are updated at a much slower pace: there is a propagation delay, ranging on average from several minutes to several hours. Clients are aware of this: first, they know that repository units can differ for a short time between replicas; second, a lock may only be obtained if the local replica has the most recent content. Thus, acquiring the lock provides a strong consistency guarantee for an individual repository unit. This is desirable since clients often fetch data prior to modifying it and writing it back. No such guarantee applies to the repository in general: replicas as a whole are never known to be completely consistent.

In the following, we distinguish lock operations from data movement. The idea is to require synchronous inter-site access only during lock related operations, while update propagation takes place in background.

3 Lock Management

There is a separate lock database kept at each replica, but the lock information of each repository unit is not replicated. Instead, it is managed at only one replica, usually the one where the unit was first created. If a lock request to this unit is issued from another replica, the request is redirected to the managing replica. This considerably simplifies the complexity of the system at the cost of lock service availability during network partitions.

The lock manager for each repository unit manages a space of monotonically increasing stamps (integers) and a lock governing the allocation of these stamps. Before updating the content of any repository unit, the caller must first acquire the lock and get a new stamp. So, for each unit, replica *i* controls:

S[i] the current stamp

The managing replica also controls:

S_L the last allocated stamp

U the current lock owner, NIL if unlocked

The system maintains the following invariants over each repository unit:

1. S_L increases monotonically
2. *S[i]* increases monotonically for all *i*
3. $S_L \geq$ *S[i]* for all *i*
4. if *U*=NIL then S_L=*S[i]* for some *i*

Users of the system perceive three basic primitives: *lock*, *unlock*, and *ship*, which maintain these invariants:

```
Lock (u: User; i: Replica) =
    IF U = NIL AND S[i] = S_L THEN
        U := u;
    ELSE Fail();
    END;
```

The *lock* operation attempts to acquire *U*. If the caller's replica (*i*) is not up-to-date, the operation must fail since a pending update might take indefinitely long to arrive.

```
Ship (u: User; i: Replica) =
    IF u = U THEN
        S_L := S_L + 1; s := S_L;
        IF Update(i) AND s > S[i] THEN
            S[i] := s;
        ELSE Fail();
        END;
    ELSE Fail();
    END;
```

The *ship* operation checks that the caller holds the lock and requests the allocation of a new stamp. The caller's replica is then updated with the new content and if all goes well, the new stamp is written to *S[i]*. During the update operation, the lock can be broken and *S[i]* and S_L can change (this is why the temporary variable *s* is necessary). *Ship* accounts for the possibility that the lock has been broken by checking that s > *S[i]*, (see *break* below). In the event of failure (*e.g.* cancellation or crash), *S[i]* remains unchanged. Note that *ship* maintains the monotonicity of stamps which ensures that invariants (1), (2) and (3) will always hold.

```
Unlock (u: User; i: Replica) =
    IF u = U AND S[i] = S_L THEN
        U := NIL;
    ELSE Fail();
    END;
```

The *unlock* operation requires that the caller holds the lock and that the caller's replica be up-to-date. This maintains invariant (4). Notice that the lock may be reacquired prior to update propagation as long as *S[i]*=S_L.

In practice, we must be able to break locks: users can forget to unlock before going on holidays; replicas with up-to-date content can be inaccessible for a long time; allocated stamps can be orphaned by failed ship operations. Invariant (4) states that the lock cannot be released until some replica possesses S_L. Thus, we need a mechanism to *break* the lock:

```
Break (i: Replica) =
    IF S[i] ≠ S_L THEN
        S_L := S_L + 1; S[i] := S_L;
    END;
    U := NIL;
```

Figure 1: Siphon Architecture

Lock/unlock requests are directed to the lock server; get/ship operations are directed to file servers; and the siphon server is responsible for data movement to and from other sites. Note that data movement is always bi-directional.

What happens when a user ships new content? First, a specialized tool updates each local sub-replica. To avoid unnecessary copying of data, only files whose timestamps or lengths differ are copied. When at least one sub-replica has written the new content to stable storage, *S[i]* can be updated at lock server. The lock server then requests the local siphon server to update all distant replicas. The siphon server gets the data from a local file server and sends it to remote sites. The siphon server at each remote site then ships the received data to its file servers.

As stated above, data movement is asynchronous: the siphon server manages a queue of pending updates to distant sites. We use a variety of techniques for making optimal use of communications lines: file timestamps are maintained to avoid unnecessary copying of data, data compression is used to double communications bandwidth, and received data is cached in stable storage to avoid retransmission in the event of line failures or system crashes. In addition, the Siphon uses the network topology to avoid copying twice the same data over the same communication lines (see Routing Updates below).

Since many machines and connections are involved in the Siphon system, we devoted special attention to the problems of crash recovery and data consistency:

- Lock server data is maintained in stable storage, using a snapshot and log technique [2].
- Since updates can cause temporary inconsistencies between sub-replicas and the lock server, all actions are recorded in stable storage so that valid state can be recovered in the event of a lock server crash.
- Each sub-replica checks periodically if it has the current content for each repository unit, and tries to remain current by fetching more recent content. (Sub-replicas can miss updates due to crashes and local network partitions.)
- The siphon server at each site periodically compares the local stamp of each repository unit with those at other sites. If the local data is out-of-date, it asks a more current replica for an update.
- A daemon computes fingerprints over the contents of each replica and checks that the replicas actually do converge. The fingerprinter actually checks data as opposed to stamps, so we learn quickly about bugs in our methods.

As with replicas, there is no guarantee that all sub-replicas will have consistent content at any given time. There is a window of inconsistency due to propagation delay. Of course, this interval is typically much larger between replicas than between sub-replicas.

5 Sharing Software with the Siphon

In practice, each repository unit is a file system directory that we call a *package*. Packages are typically self-contained software or documentation components which *export* software interfaces, libraries, executables, and documents for general use. Packages are usually authored and maintained by an individual or a small number of people. Typical sizes range from .01 to 40 MBytes. We have roughly 1000 such packages totaling 2.5 GBytes. Of this total, about 40 MBytes of data propagate through the system per day.

Users modify packages following a *lock/get/modify/ship/unlock* paradigm. The user first locks and gets a copy of the package in a private directory. This is usually done through a single command, *getpackage*, which locks the package and gets its content from an up-to-date file server (an up-to-date file server must exist at local site, otherwise the lock couldn't have been acquired). Then, the user modifies and tests his private version of the package. Of course, the "public" package stored in the repository is not changed during this phase. When the user is satisfied with his changes, he uses the *shippackage* command to ship the new package content from his private directory to the repository, publicizing it to all users at all sites. Finally, the user unlocks the package, allowing other people to modify it. Note that this is not a versioning system: old package content is destroyed and replaced by the new one.

Once the package has been distributed to a site, the Siphon provides an export facility for making selected files available to end users. In most operating system environments, the system management task of installing new software is typically performed by a small set of authorized people. Often the same task needs to be performed on many different file systems. This can be tedious and error prone, even when the systems to be updated are not geographically distant. The Siphon export mechanism makes it possible for a non-privileged user to install, in a single action, new software quickly and reliably over a large network. Because the Siphon itself is reliable in the face of network failures, the export process is as well.

When a user ships a package, he indicates in the *shippackage* command which files should be exported, and, for each such file, the name of the directory to export it to. An export directory is typically a well known location suitable for inclusion on a search path. For example, an export directory containing executables could be included on a user's command shell search path. A compiler might search an ordered set of export directories in order to locate an include file.

In practice, a symbolic link is written in the export directory to each exported file, thereby making it visible to end users. The Siphon then carries sufficient information to recreate each export link at each replica. Thus, when a user at a site ships a package which exports an executable, all users at all sites will soon see the new version of the executable. Beware of errors!

This export mechanism is not mandatory: some packages don't have exported files. One could envision a siphon system without any exported files at all. But for our software development environment, we found this automatic installation feature invaluable. Newly released

software interfaces and libraries can and do become available for sharing instantaneously. No system administrator action is required. Moreover, the change is propagated around the world in a reliable fashion. Of course, this requires a certain trust in the user community, since a random user, 10,000 miles away, can change your computing environment! Although we haven't found it necessary, it would be simple to modify the Siphon system to restrict updates to the repository with access control lists.

6 Selective Sharing

Presently, the package data base is huge. Since not all sites are interested in all packages, the Siphon allows selective sharing of packages, thus making it possible to distribute different sets of software to different sites. This selective sharing is provided at the level of groups of related packages, called *sub-repositories*. Sub-repositories are organized in a tree structure, which reflects the file system structure used to store them: to each sub-repository there corresponds a file system directory, which in turn contains the related packages directories. The relationship between packages in a sub-repository can be of any kind (packages written in the same language, targeted for the same system, or related to a given domain: graphics, mathematics, etc). For example, we could have sub-repositories */proj/graphics* and */proj/maths* (by convention, */proj* is the name of the global repository). If we also wish to distinguish between vax and mips (DECstation) architectures, we could have */proj/graphics/vax*, */proj/graphics/mips*, etc.

Each sub-repository can be shared by all replicas, or by a subset, or by only one replica. This multiple repository scheme also applies to sub-replicas. Thus, file server disk sizes can be tuned according to their real use.

In order to minimize network traffic, the siphon server implementation sends a package update only to sites sharing the package sub-repository. For this purpose, each siphon server periodically interrogates the other sites in order to know which sub-repositories they share.

Selective sharing is not provided at the level of individual packages. There are several reasons for this. First, we want system administrators to determine the sub-repository structure and sharing, and let users choose sub-repositories for their packages. For users, choosing a sub-repository for a package, *i.e.* finding a group of related packages, should be straightforward, while finding which sites and which sub-replicas may be interested by a package can be tedious. Second, site interests can change over time, new sites can be added, and we don't want to place on users the administrative burden of changing the sharing parameters of their packages. Third, related packages are sometimes strongly dependent. For example, if an application package depends upon a library in another package, it makes no sense to share the first package without sharing the second. This kind of dependencies can be reflected with sub-repositories, but not with individual package sharing.

Because of these "hidden" dependencies, defining a sub-repository structure is not at all obvious, especially when starting with a flat packages structure, as in our case. It is easier if the sub-repository structure is defined from the beginning.

7 Routing Updates

Logically, all sites are fully connected: each site can send data to or receive data from any other site. In practice, the Siphon uses direct connections only for lock operations, since lock operation data is small and always targeted to one site only. For update data, this is not the case: if we choose to send updates directly from the site that issues a ship to all other sites, we could put an unnecessary load on the network. Suppose for example that sites A, B and C are connected linearly: A - B - C and that an update has to be sent from A. If we send the update directly to B and C, data will be uselessly copied twice on the A-B path, thereby slowing down the transfer. Instead, the Siphon sends the update to B and asks B to forward it to C.

In order to do this, an *update propagation route* is computed for each update. This route is sent along with the update data. After receiving the update, the receiving site forwards it according to this route. This route is computed from a weighted graph describing the physical network, where nodes are sites and edges are physical communication lines, the weight of each edge being inversely proportional to the bandwidth of the corresponding line. From this graph, the Siphon computes the minimal cost path for each remote site, and merges these paths to form the update route. This ensures that each communication line is used only once for the same data.

In the current implementation, a site cannot forward a package if it doesn't share it. Thus, the update route for a package must avoid all uninterested sites. It is obtained by using a graph derived from the complete graph above.

Routing of updates supposes knowledge about the network topology. This is easy when using a private network of dedicated lines, but more difficult, sometimes impossible, when using a general purpose network. In the latter case, we can of course assume that all sites are physically connected and use a fully connected graph of equally weighted edges. But it is worth trying to reflect at least part of the network topology in the graph to avoid unnecessary data transfers.

Note that routes are fixed and cannot change according to the network load or in case of network partitions. The latter case is handled differently. As stated before, the siphon server at each site periodically compares the local stamp of each package with those at other sites. If the package is out-of-date, the siphon server asks, after a certain delay, the nearest up-to-date site to send an update. The delay is there to avoid unnecessary resend requests, since the normal update propagation mechanism can take some time.

8 Implementation - Installation - Management

The Siphon system has been in operational use since January of 1989. It is implemented in Modula 2+ [4] and runs under the SRC Topaz [3] environment and several UNIX[1] variants. All communications primitives are implemented as remote procedure calls [1] and have been

[1] UNIX is a trademark of AT&T Bell Laboratories

demonstrated to work using either IP or DECnet transport protocols. Currently, our slowest network paths run at 56 Kbit/sec, with one transoceanic satellite hop.

In practice, the Siphon works pretty much automatically. There have been very few operational problems although network partitions, in the form of broken overseas telecommunications lines, have caused us considerable inconvenience. In our particular system, the most important requirement for communications has turned out to be availability, not latency or bandwidth. While 56 Kbit/sec seems slow in the context of modern networks, a third of this bandwidth would have been adequate for our purposes.

The Siphon is managed through various kinds of administrative tools. Some tools give statistics such as update frequency, line use, line throughput, effective bandwidth, etc. Some tools are provided to deal with problems, for example, forcing immediate update of a replica or sub-replica, changing an entry in the lock database. As stated above, we made very little use of these emergency procedures. Other tools are intended for normal operations, for example, creating/deleting a sub-repository, changing the sharing of a sub-repository, creating/deleting replicas and sub-replicas.

All servers involved (siphon server, lock server, sub-replica servers) get their configuration information from a unique file, which contains the name of local and remote servers, and the graph describing the network topology. Creating or deleting a sub-replica is straightforward: simply change the configuration file, without stopping the other sub-replicas or the siphon server. Adding a replica is more cumbersome: recently, a third replica has been added to our system. It took about two days. It was encouraging to find that this required no changes to the existing implementation.

Installing a siphon system between a new set of sites is a relatively painless process, although it can be slow if large amounts of data need to be moved. (To be frank, we haven't worked much in this area and there is room for improvement.) However, taking advantage of the Siphon's full functionality is more difficult since it affects many aspects of system organization (directory structure, inter-machine replication, disk-space allocation). This is especially true for the export file facility, which also impacts the programming methodology. However, in our experience, the benefits in terms of system administration and added user functionality was well worth the effort.

9 Future Work

It was initially our hope to treat source files and derived files differently in the shared SRC-PRL repository. Derived files, which are often large, might well be regenerated rather than copied in bulk. This proves difficult as long as there is no history of previous package content. In our current system, modifications replace, rather than augment, the existing state.

Suppose, for example, that a package contains a critical system library and that a pending update to this library contains an interface change. At the point the change is made, dependent code requires at least recompilation and possibly source modification, so let's suppose further

that the software contained in all such packages can be quickly updated as well. Even if this were practical, it would pose an additional problem in that update propagation must now be subject to complex ordering constraints.

A Siphon-like system integrated with a conventional source control apparatus (*e.g.* RCS [6]) might provide version history, but would lack any coherent history about how packages interact. Although rebuilding of derived files might be possible with the knowledge of which versions of each package combine to form a complete system, it's hard to see how this could be made automatic.

A better solution would be to integrate the Siphon with a software development environment that not only implements a revision history for each package but also maintains precise information about the structure and dependencies inherent in all derived files. Such a system, is currently being developed at SRC. This system will support repository replicas in much the same way as does the existing Siphon. Since package versions will be immutable, even fewer constraints exist on the structure and distribution of the lock database. The resulting system should provide the power to fork development paths easily, and to rebuild arbitrary derived files from scratch, at any participating replica.

10 Conclusions

We believe that the kind of loose consistency provided by the Siphon works well for managing multi-site software development. Since modifications to single components are often controlled by individuals or by small, co-resident groups, the percentage of lock operations which are local to the managing replica is quite high. Nevertheless, in those cases where off-site modification are required, the functionality is available so long as no network partition exists. Moreover, the greatest benefit of the system comes in the form of low latency read access at distant replicas. At these replicas, as long as a strong consistency guarantee is not required, repository reads can perform at file system speed. Furthermore, new content appears in an automatic and timely fashion and network partitions don't constrain visibility of previously propagated updates.

Our experience with the Siphon has been a positive one. It has enabled SRC and PRL to achieve a much higher degree of synergy than would have been possible with conventional tools. Researchers can collaborate on software artifacts with relative ease, and innovations at either lab appear promptly at the other. The image of a single, shared repository has been achieved.

References

1. Birrell, A.D. and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst. 2*, 1 (Feb. 1984)

2. Birrell, A.D. et al. A Simple and Efficient Implementation for Small Databases. *Proceedings of the Eleventh Symposium on Operating System Principles*, ACM, New York, (Nov. 1987)

3. McJones, P.R. and G.F. Swart. Evolving the UNIX System Interface to Support Multi-threaded Programs. Research Report 21. Digital Systems Research Center. (Sept. 1987)

4. Rovner, P. Extending Modula-2 to Build Large, Integrated Systems. *IEEE Software 3*, 6 (Nov. 1986)

5. Schroeder, M.D. et al. Experience with Grapevine. *ACM Trans. Comput. Syst. 2*, 1 (Feb. 1984)

6. Tichy, W.F. Design, Implementation, and Evaluation of a Revision Control System. *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, (Sept. 1982)

PRL Research Reports

The following documents may be ordered by regular mail from:

Librarian – Research Reports
Digital Equipment Corporation
Paris Research Laboratory
85, avenue Victor Hugo
92563 Rueil-Malmaison Cedex
France.

It is also possible to obtain them by electronic mail. For more information, send a message whose subject line is `help` to `doc-server@prl.dec.com` or, from within Digital, to `decprl::doc-server`.

Research Report 1: *Incremental Computation of Planar Maps*. Michel Gangnet, Jean-Claude Hervé, Thierry Pudet, and Jean-Manuel Van Thong. May 1989.

Research Report 2: *BigNum: A Portable and Efficient Package for Arbitrary-Precision Arithmetic*. Bernard Serpette, Jean Vuillemin, and Jean-Claude Hervé. May, 1989.

Research Report 3: *Introduction to Programmable Active Memories*. Patrice Bertin, Didier Roncin, and Jean Vuillemin. June 1989.

Research Report 4: *Compiling Pattern Matching by Term Decomposition*. Laurence Puel and Ascánder Suárez. January 1990.

Research Report 5: *The WAM: A (Real) Tutorial*. Hassan Aït-Kaci. January 1990.

Research Report 6: *Binary Periodic Synchronizing Sequences*. Marcin Skubiszewski. May 1991.

Research Report 7: *The Siphon: Managing Distant Replicated Repositories*. Francis J. Prusker and Edward P. Wobber. May 91.

Research Report 8: *Constructive Logics. Part I: A Tutorial on Proof Systems and Typed λ-Calculi*. Jean Gallier. May 1991.

Research Report 9: *Constructive Logics. Part II: Linear Logic and Proof Nets*. Jean Gallier. May 1991.

Research Report 10: *Pattern Matching in Order-Sorted Languages*. Delia Kesner. May 1991.

Research Report 11: *Towards a Meaning of LIFE*. Hassan Aït-Kaci and Andreas Podelski. May 1991.

Research Report 12: *Residuation and Guarded Rules for Constraint Logic Programming.* Gert Smolka. May 1991.

Research Report 13: *Functions as Passive Constraints in LIFE*. Hassan Aït-Kaci and Andreas Podelski. May 1991.