Lock-Free Decentralized Storage for Transactional Upgrade Rollback*

Boris Mejías, Gustavo Gutiérrez, Peter Van Roy
Université catholique de Louvain
firstname.lastname@uclouvain.be

John Thomson, Paulo Trezentos
Caixa Magica Software
firstname.lastname@caixamagica.pt

Abstract

Installing and upgrading software may introduce conflicts and errors into a system. Transactional Rollback allows the system to return back to a previous, stable and known state. However, to perform such a rollback, it is necessary to store a large amount of information including configuration and installation logs, as well as different versions of software packages. Nevertheless, much of this information is common to several users using the same software and performing the same operations. We can reduce the total amount of storage by having a decentralized architecture using a Distributed Hash Table (DHT) to localise shared resources. We propose a lock-free key/value-set protocol to add and remove data from the DHT. The lock-free protocol is not limited to transactional rollback, and it can be used by other applications that also need value-sets as part of their stored data.

1. Introduction

Restoring system software to a state at a previous timepoint is something that in computer science can be pursued at different levels. One can undo an action at an application level, can restore personal files or can rollback to a previous version of an application just installed. Our work concerns the latter: rolling-back between various states at a package level of granularity. Although our focus is in GNU/Linux systems, the findings can be extended to other operating systems like MacOS, BSD or even Windows.

Software distribution in the free and open source software community is mainly done by servers and mirrors hosting a large repository of packages which are constantly updated. As we will describe in more detail in Section 3, there is a trend towards using the resources of the community users to help with software distribution, creating a peer-to-peer network for decentralized storage. Based on that idea, we identify the possibility of storing the information that is needed to perform upgrade rollbacks, reducing the amount of redundant information within the community, and improving the performance of the software managing system.

To index the decentralized storage we will use a DHT. To provide consistency and robustness, we will use transactional protocols based on the Paxos-consensus algorithm. To improve performance, we propose to complement the traditional key/value pair abstraction for DHT, with key/-value sets, which are the main abstraction provided by OpenDHT [9]. The operations to replicate the value-sets have been designed with lock-free protocols to overcome the problems indemnic to the dynamic nature of peer-to-peer systems, as distributed locks are an important source of problems in distributed programming.

2. Transactional Upgrade Rollback

Concerning software management, several tools exist that provide rollback features in various guises:

- apt-rpm[1]: is a port of apt tools to RPM Package Management (RPM) based systems. Rollback was introduced as a branch to apt-rpm by Caixa Mágica. Rollback works by hooking into apt-rpm and logging the changes to the system into a sqlite database. Configuration files are maintained by the suite and as long as the old package version is preserved with the log, rollback can be performed.
- Nexenta OS rollback: uses ZFS as the basis for providing rollback features. Using a modified version of apt, apt-clone, it combines the snapshot ability of ZFS with the package installer system allowing the operating system to create new boot points each time a modification is performed. The snapshot mechanism captures the state of files on the system at a given time, on modification of a package, that is then preserved and can be recalled by the user or the system at a later date.
- NixOS: uses a unique mechanism that differs greatly from other rollback techniques. It relies on a sub-

*The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under Grant agreement n 214898. The Mancoosi Project.

[1] http://aptrpm.caixamagica.pt/

stantial shift in how the system performs operations, by using a purely functional approach for applications. Configuration changes are not stateful and any modifications to packages on the system, do not modify any other parts and as such rollback is performed by using the previous set of files.

These and other tools use different approaches to implement rollback: pure functional, file system snapshot or package association. However these approaches have several limitations that are blocking the wide-scale uptake and use of these tools.

In the case of a pure functional approach as developed for NixOS the main problem is that it is disruptive and the approach breaks compliance with Linux Standard Base (LSB) and the system differs from most other standard GNU/Linux based operating systems. A large issue pertains to maintainer scripts and most of the time actions performed by them have an impact outside of the installed package domain and affect the system or other packages. As part of the work into Deliverable 3.2 for the MANCOOSI project [2] we investigated the popularity of these scripts and it was clear that hand-written scripts used in packages are a minority.

Using a Domain Specific Language (DSL) to express the actions performed within the maintainer scripts provide us with a powerful tool to:

- Detect if the rollback operation is possible or not. This can be done in an explicit or implicit way.
- Serialize the steps performed during the installation, avoiding evaluating environment variables where conditions might change later.
- Provide the reverse functions of each statement or group of statements named as template. In this way if an installation fails part way through, these operations can be rolled back, by performing the associated reverse functions.
- Log the serialized instructions to later provide a rollback path.

The log mechanism works in combination with modified maintainer script files. RPM at certain stages executes sets of commands in a particular order around the installation of files and provide a way for the maintainers to get the information that cannot be known a-priori and to integrate with the end user's system. RPM runs these scripts by passing them to the shell but only receives the error/success code. RPM thus does not know where in a maintainer script it has failed and for what reason. It therefore has no ability to rollback the changes performed by the maintainer scripts. By adding serialised DSL elements that interact with the log we are able to see where the maintainer script has failed and if there are associated roll-back commands, then they can be performed in reverse order either at the time, or at a later point. As long as the logs are kept consistent, they can be used to perform the rollback. The new logs contain pointers to the rollback mechanism produced by Caixa Mágica for apt-rpm and add additional information such as the DSL and rollback commands. The DSL commands provide a functional 'why' for maintainer scripts that detail the 'how'.

For assuring the correct rollback, using this mechanism, from T to $T-1$, one must assure the presence of the following elements at the instance of rollback:

1. The target version of the package $T-1$
2. Package T maintainer scripts with reverse statements
3. The log of the transaction

The above elements can be stored in a peer-to-peer network. In fact, both package version $T-1$ and maintainer scripts are common between users and can benefit from a peer-to-peer strategy.

The transaction logs are not shared between users but benefit in being in the network for fault-tolerance reasons. Since the size of the logs are quite small we believe the end users might be interested in storing other users' transaction logs, in a secure way, to benefit from the same service.

If the system is completely lost for some reason, a user can rebuild the whole package system based on the information stored on the network. User files not related to packages would not be stored on the network. The only user centric data stored is that of the log and if this is stored in an encrypted way and the correct policies are applied within the peer-to-peer network then there should be no problems with this approach. A user-selectable option for sensitive data such as the log could determine whether or not it is uploaded. This is a disruptive approach but it is feasible to implement using the DSL implementation and the proposed lock-free protocol.

3. Peer-to-peer Software-Package Distribution

Motivated by the success of peer-to-peer file-sharing applications, several approaches have targeted the goal of making the distribution of software packages use a peer-to-peer system, so as to reduce the load on servers and mirrors, and increase download speed for the clients. In this section we discuss some of them, and we describe our proposal to include support for upgrade-rollback.

The EDOS Project [1] uses a peer-to-peer approach for package distribution and uses metadata information for content searching. Basically they provide an infrastructure for solving several problems: resource handling from the distribution side (e.g. resource sharing, load balancing), an

[2] http://www.mancoosi.org/reports/d3.2.pdf

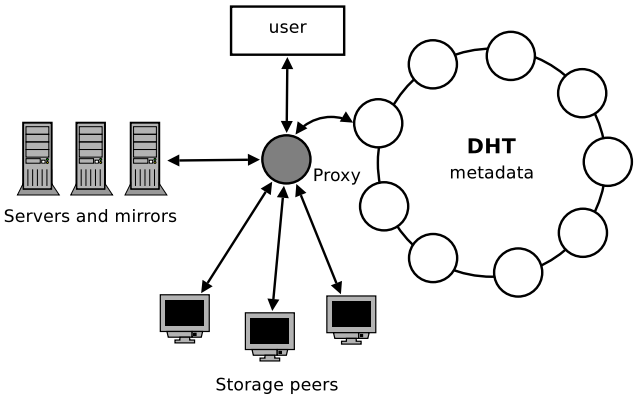

Figure 1. apt-p2p's architecture

information system based on the metadata extracted and a subscription/notification based update system. Three granularities are used in the system: package, utility and collection (e.g. to represent a full Linux distribution). For a common scenario like a software update, the system generates the metadata according to the granularity and indexes it in the distributed system. With this information, all the peers (according to their role) can start serving the update.

The system consists of three different types of peers: publishers, mirrors and clients. A publisher peer is able to add new content into the system and to manage client subscriptions for the upgrade system. A mirror peer is able to download and is mostly targeted at data replication but additionally it is 'trusted' for use for indexing purposes. Finally, the clients are only allowed to download and subscribe, and are considered 'not trusted' for indexing purposes.

In Debian-based Linux distributions[3], we find `apt-p2p` [3], which transparently integrates with the official packaging tool: `apt`. Figure 1 describes `apt-p2p`'s architecture. The approach uses a proxy to communicate directly with the servers and mirrors hosting the software-packages. The proxy also connects to a distributed hash table (DHT) to find other peers hosting the same packages the client is interested in. If the packages can be found on other peers, the proxy does not request them from the mirrors, saving them some upload bandwidth, spreading that cost within the community.

Peers participating in the DHT and that are hosting packages are also running an instance of `apt-p2p`. They can also download packages from other peers. Some peers may participate only as DHT nodes, or only as package storage. This can depend on their uptime and whether they are behind a NAT or not. We find this architecture very simple and effective. By using a DHT, it overcomes the problems of using torrents for package distribution.

[3]Debian, Ubuntu, Knoppix and others.

Torrents do not work very well for package distribution because of the way software evolves in the free and open source community. Distributions are too large to put in one torrent and means that users would have to download a lot of software they would not use. Another reason is that periodically there are new versions for packages, resulting in a high rate of recreation of torrents, dividing the seeders even when they share several packages in common. On the other hand, if every package is put in its own torrent, around 80% of them would have a size which is less than the minimal download chunk (512KB), making the use torrent suboptimal. DebTorrent [4] a predecessor of `apt-p2p`, tries to overcome these problems, and it implies changes in the way software is currently distributed.

The use of the DHT not only overcomes torrent's problem for distributing software but it also offers many possibilities to store more data which we will use to enhance our rollback mechanism. Currently, `apt-p2p` stores in the DHT a list of peers hosting each package. When a user downloads the package, it updates the list putting itself as one of the hosting peers. In our approach, when the user performs the installation upgrade, it establishes the rollback transaction using the DSL, as described in Section 2. The user logs the transaction on their own machine, and stores meta-data about it in the DHT, so it becomes available to other peers, creating a decentralized database of rollback-transactions.

3.1. Difference between packages

Another part of the installation will compare both versions of a software package, and it will establish the difference between them. To establish the difference, we can use dedicated software for this task, namely Bsdiff [5] `http://www.daemonology.net/bsdiff/` and Courgette [6]. Both tools are concerned with finding the differences between two binary files and make a patch between them. An important aspect on which both tools rely on is that even minor changes in the source code will produce completely different binaries but the part with the most changes is at the creation of pointers. To avoid producing a big patch a common approach is to create an external file and to index the changed addresses and replace them by labels. By doing this the difference is taken on real changes and not concerned with changes introduced by the compiler when generating the binary file from similar (i.e. with minor changes) source code. According to Courgette, the part of a binary file that remains unchanged is about 80%.

The main difference between both tools is that Courgette uses a disassembler as part of a preprocessing step. By doing that, it is able to store some *guess* that improves the pro-

[4]`http://debtorrent.alioth.debian.org`
[5]`http://www.daemonology.net/bsdiff/`
[6]`http://www.chromium.org/developers/design-documents/software-updates-courgette`

cess of creating a patch. Courgette is currently used to distribute updates for the Google Chrome web browser where Bsdiff was used before. In the worst case, when changes are huge, the performance of Courgette will be similar to Bsdiff. Both tools work at the file level, this is, they are not concerned with directories or packages. An important aspect to be discussed is how to get a better abstraction level for the scenarios described here. For example, software packages contain binary and text files. For both types there exists a solution but integrating them in a transparent way will lead to an abstraction at the package level. Metadata about *diffs* is stored in the DHT, so that patches and software packages can be retrieved from other peers.

3.2. Combining upgrade and rollback

We propose to use the existing architecture of `apt-p2p` to combine information about upgrade and rollback. Ideally, we would need to store only meta-data about the difference between versions of software packages, and the patches will be determined dynamically as they are needed. Working at the granularity of files, we assume the following situation. Peers p and q have package foo-2.0, which is composed of files $f1$, $f2$ and $f3$. Peer p upgrades to foo-2.1. In the process of upgrading, peer p determines which files differ between the two versions, and it stores the corresponding meta-data in the DHT. Let us say that only $f3$ is different between the two versions. That information will be stored in the DHT. Along with the diff, peer p register itself as host of foo-2.1, and it also stores in the DHT the meta-data concerning the DSL rollback-transactions.

If p needs to rollback to foo-2.0, at a certain point it will need to download that software. Instead of downloading the whole package, which is unnecessary considering that files $f1$ and $f2$ are unmodified, it will find out that it only needs file $f3$, and that it can be retrieved from peer q. Similarly, when peer q wants to upgrade to foo-2.1, it will just need to download file $f3$ from peer p, or any other peer hosting it. In this manner, both upgrade and rollback benefit from the combined approach. To store the list of peers in every data item of the DHT, we will use a data structure name key/value-set, which can be modified without the need of distributed locks. The protocol is described in the following section.

4. Decentralized Storage

Structured overlay networks represent the third generation of peer-to-peer systems. Inspired mainly by Chord [11] and Kademlia [5] among others, the DHT became one of the main abstractions for decentralized storage. Different replication strategies were designed to provide fault tolerance. In Kademlia, which is the DHT used by `apt-p2p`, every item is replicated six times using the direct neighbours of each responsible peer. If a given peer p is responsible for key k, the key/value pair (k, v) is stored in p, it is also stored in the three preceeding peers of the circular address space, and then in three succeeding peers. This strategy has the disadvantage that the system always needs to go first through the peer responsible for a key to find the replicas, creating congestion.

We prefer the symmetric replication [4] strategy implemented in systems such as Scalaris [10] and Beernet [6]. To determine the f replicas of a key/value pair (k, v), the circular address space is divided into f partitions, where the responsible peer is chosen by the formula:

$$(k + i * \lfloor N/f \rfloor) \; mod \; N$$

where N is the size of the address space, f is the replication factor, and i is a natural number going from 1 to f.

4.1. Transactional Replicated DHT

Replicating each key/value pair independently is not enough. Systems often need to modify several key/value pairs, also called items, with the property that all changes are committed to the replicas, and if one of the updates fails, all of them get aborted too. In other words, transactional access to the DHT is required. Note that in this section, we mean "transaction" as an atomic modification of several items on the decentralized store, and not the same kind of transaction we described in Section 2.

The basic idea of a transaction is that one peer behaves as the *transaction manager* (TM). Every peer holding a replica of each item involved in the transaction is a *transaction participant* (TP). The most basic protocol to run the distributed transaction is a Two-Phase Commit. It is used by several relational databases, using a hierarchical organization of the nodes. The protocol is defined as follows: the TM request to all TPs to lock the items in order to perform the update. The participant will vote *abort* or *commit* according to whether the lock is free to be taken or not. When the TM collects all votes, it can decide if the update is committed to all replicas or not. Two-phase commit strongly relies on the TM, making it infeasible for peer-to-peer systems, where peers join and leave the network very often. If the TM leaves the network before the transaction is committed, the items remain locked forever.

Atomic transactional commit has been achieved successfully on peer-to-peer networks by adapting a Paxos Consensus algorithm [7]. The Paxos-based protocol improves two-phase commit in two main ways: it only needs to reach an agreement with the majority of the TPs to commit a transaction, and it uses replicated transaction managers (rTMs) to guarantee the resilience of the TM. The algorithm works as follows: when the TM is ready to request the locks from the TPs, it first registers the rTMs setting them up to collect the votes. The TPs will send their votes not only to the TM, but also to all rTMs. The rTMs will acknowledge each other upon receipt of votes. When a majority of votes is reached,

the TM is ready to take a decision. Since votes are sent to all rTMs, if the TM fails at this moment, any rTM is ready to take over because it has also collected the votes.

The Paxos-based protocol has been successfully implemented in Scalaris [10] to build a fully decentralized version of wikipedia [8]. The protocol works very well for asynchronous collaborative applications. In Beernet [6], the protocol has been extended to support eager locking, making it feasible to build synchronous collaborative applications. In both cases, locks are the only way to guarantee atomicity, concurrency control and strong consistency. Unfortunately, locks are not the best abstraction for distributed systems and it is highly desirable to avoid them whenever possible. The importance of lock-free protocols for cloud computing has also been identified by the research community [2]. We propose a lock-free key/value-set abstraction that allows us to develop an important part of the package management system with decentralized storage.

5. Lock-Free Key/Value-Set

We want to use the transactional DHT to store the set of peers hosting software packages, rollback transactional logs, and to store meta-data about the difference between packages' versions. In the particular case of storing a set of peers, it is unnecessary to have strong consistency, because we are only interested in being able to contact some peers to retrieve a package. We just need for the set to be eventually consistent, to prevent contacting peers that are already removed from the set, and to be able to eventually contact peers added to the set.

Let us assume we want to store the set of peers hosting package foo-2.1 using a key/value pair. The key would be foo-2.1, and the value would be the whole set. When two users want to add themselves as new hosts simultaneously, only one of them would succeed acquiring the lock for the entire set. However, It is unnecessary to lock the whole set if two different values will be added to it. Therefore, we propose an *add(k, v)* operation that will add the value to the majority of the TPs without taking any lock. Every TP will eventually obtain all the values added to the set. By reading from the majority with the operation *readSet(k)*, strong consistency can be guaranteed. Values can be removed with *remove(k, v)*.

Table 1 shows several examples of adding, reading and removing values from a set under key k. Peers p, q and r are the TPs containing replicas of the set. Column t shows time evolution, and the column TPs indicates which peers were successfully contacted to perform each given operation.

We can observe that at time t_2, even when the *readSet(k)* operation can only contact peers q and r, and that none of them has the entire set. It is possible to naively construct it by just adding all the found elements. However, this only works when *add* operations were performed. Observing

Table 1. Deducing the value set

t	Operation	TPs	p	q	r
t_0	add(k, a)	{p, q}	{a}	{a}	ϕ
t_1	add(k, b)	{p, r}	{a, b}	{a}	{b}
t_2	readSet(k)	{q, r}	→ {a, b}		
t_3	add(k, c)	{p, q, r}	{a, b, c}	{a, c}	{b, c}
t_4	remove(k, c)	{p, q}	{a, b}	{a}	{b, c}
t_5	readSet(k)	{q, r}	→ {a, b} or {a, b, c}?		

Table 2. Identifying and storing operations

t	Operation	TPs	p	q	r
t_0	i: add(k, a)	{p, q}	(i)	(i)	ϕ
t_1	j: add(k, c)	{p, q, r}	(i, j)	(i, j)	(j)
t_2	j': remove(k, c)	{p, q}	(i, j, j')	(i, j, j')	(j)
t_3	readSet(k)	{q, r}	→ {a}		

times t_3 and t_4, we can see that peer r missed the removal of value c, therefore it is impossible to determine at time t_5 what the real set is even if all the peers would have been contacted. The problem is basically that peers p and q do not keep track of the removal of value c, until all TPs have acknowledged it.

Instead of only storing the values, the TM assigns an identifier to each operation, and every TP stores the operation together with its id. When *readSet(k)* contacts the majority, the set can be reconstructed from the operations. Table 2 shows how ids are assigned to operations and how it is possible to reconstruct the whole set by contacting only the majority of TPs at time t_3.

Figure 2 describes the lock-free protocol for key/value-sets. Similar to the Paxos-consensus algorithm, the client sends one or more operations within a transaction to a TM. The TM creates an id for each operation, and registers the rTMs to start collecting votes from the TPs. Each TP is asked to vote on the operation for which is involved. The only reason to reject an addition is that a concurrent transaction is trying to add the same value. Because operations are identified, it is necessary to only accept one. The only reason to reject a removal, is that the value is not yet stored. There is a partial order between addition and removals for each value. This is to guarantee the reconstruction at the time of reading. Once TPs get the decision from the TM, they acknowledge with each other the new operation. This is to guarantee that if a TP misses the transaction, it will eventually receive the operation from the other TPs.

When two or more transactions try to concurrently add the same value to the same set, at least one of the TMs will not obtain a majority of successful votes. Instead of abort-

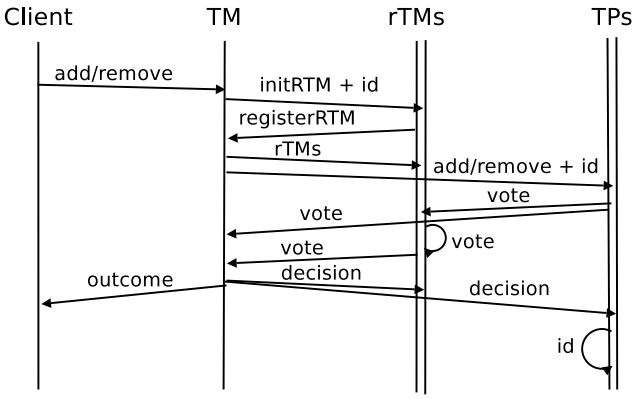

Figure 2. Lock-free key/value-sets protocol

ing the transaction as in Paxos, the TMs will wait a random time to generate a new id and retry the operation. If the value has been finally added by another transaction, the TM will receive votes *duplicated* from the TPs. The means that the decision can be discarded, and the client is notified that the addition was successful.

5.1. Using Key/Value-Sets

The following pseudo code shows how DHT's key/value-sets get integrated with the process of installing a package. First, it is necessary to find out the dependencies of the package. Then, for each package that needs to be installed, we obtain the list of peers hosting the package by reading the set associated to it with the call `dht.readSet(p)`. Once packages are downloaded, the peer registers itself as host for each package with `dht.add(p,self)`. Then, packages can be installed according to what we have described in Section 2. When downloaded packages are removed from the cache, which is not shown here, the operation `dht.remove(p,sef)` is used to unregister the peer from the set of hosting peers.

```
procedure prepare_installation(pkg)
    deps = get_dependencies(pkg);
    deps += pkg;
    for p in deps do
        peers = dht.readSet(p); // read set
        download(p, peers);
    end
    for p in deps do
        dht.add(p, self); // add to set
    end
    install(deps);
end
```

6. Conclusions

We have described the challenges of providing transactional upgrade rollback in software management, specificaly in the free and open source community. We have identified the possibilities of using decentralized storage to take advantage of the duplicated information installed on end-users' machines. We use a DHT to index the data which can be used for rollback and upgrading. This system results in the users getting the data faster, and reducing the congestion on distribution mirrors.

To provide an efficient and consistent DHT, we propose using transactional commits based on the Paxos-consensus algorithm, on top of symmetric replication. Aware of the problems of distributed locks, we present a lock-free protocol for key/value-sets that allows us to develop an important part of the package managing system with decentralized storage. The protocol is not limited to the system we have described in this paper, and it can be used by other systems that work with value-sets.

References

[1] S. Abiteboul, I. Dar, R. Pop, G. Vasile, and D. Vodislav. Edos distribution system: a p2p architecture for open-source content dissemination. In J. Feller, B. Fitzgerald, W. Scacchi, and A. Sillitti, editors, *OSS*, volume 234 of *IFIP*, pages 209–215. Springer, 2007.

[2] K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.

[3] C. Dale and J. Liu. apt-p2p: A peer-to-peer distribution system for software package releases and updates. In *IEEE INFOCOM*, Rio de Janeiro, Brazil, April 2009.

[4] A. Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD thesis, KTH –- Royal Institute of Technology, Stockholm, Sweden, dec 2006.

[5] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric, 2002.

[6] B. Mejías and P. Van Roy. Beernet: Building self-managing decentralized systems with replicated transactional storage. *IJARAS: International Journal of Adaptive, Resilient, and Autonomic Systems*, 2010. To appear.

[7] M. Moser and S. Haridi. Atomic commitment in transactional dhts. In *Proceedings of the CoreGRID Symposium*. Springer, 2007.

[8] S. Plantikow, A. Reinefeld, and F. Schintke. Transactions for distributed wikis on structured overlays. In *Managing Virtualization of Networks and Services*, pages 256–267. 2007.

[9] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. Opendht: A public dht service and its uses, 2005.

[10] T. Schütt, F. Schintke, and A. Reinefeld. Scalaris: reliable transactional p2p key/value store. In *ERLANG '08: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pages 41–48, New York, NY, USA, 2008. ACM.

[11] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.