Beernet: Building Self-Managing Decentralized Systems with Replicated Transactional Storage

Boris Mejías and Peter Van Roy
Département d'ingénierie informatique
Université catholique de Louvain, Belgium
{firstname.lastname}@uclouvain.be

Distributed systems with a centralized architecture present the well known problems of single point of failure and single point of congestion. Therefore, they do not scale. Decentralized systems, especially as peer-to-peer networks, are gaining popularity because they scale well, and they do not need a server to work. However, their complexity is higher due to the lack of a single point of control and synchronization, and because consistent decentralized storage is difficult to maintain when data constantly evolves. Self-management appears as a way of handling this higher complexity. We present a decentralized system built with a structured overlay network which is self-organized and self-healing, providing a transactional replicated storage for small or large scale systems.

Introduction

There are many technological and social factors that make peer-to-peer systems a popular way of conceiving distributed systems nowadays. From the technological point of view, the increment of network bandwidth and computing power has definitely made an impact on distributed systems which are becoming larger, more complex and therefore, difficult to manage. Although classical client-server architecture provides a simple management scheme with centralized control of the whole system, it does not scale because the server becomes a point of congestion and a single point of failure. If the server fails, the whole system collapses.

The key to deal with the complexity of large-scale distributed systems is to make it decentralized and self-managing. Peer-to-peer networks, and especially in their form of structured overlays, offer a fully decentralized architecture which is self-organizing and self-healing. These properties are very important to build systems that are more complex than file-sharing, which is currently the most common use of peer-to-peer. Despite the nice design of many existing structured overlay networks, many of them present problems when they are implemented in real-case scenarios. The problems arise due to basic issues in distributed computing such as partial failure, imperfect failure detection and non-transitive connectivity.

The key issue in distributed programming is partial failure. It is what makes distributed programming different from concurrent programming. This is why we would like to quote Leslie Lamport and his definition of a distributed system:

> "A distributed system is one in which the failure of a computer you did not even know it existed can render your own computer unusable"

It does not matter how much transparency can be provided in distributed programming, it will always be broken by partial failure. This is not particularly bad, but it means that we need to take failures very seriously, understanding that perfect failure detection is unfeasible in Internet style networks, and that a failure does not mean only the crash of a process, but also a broken link of communication between two processes, implying non-transitive networks. Because of failures, we cannot trust the stability of the whole system to a single node, or to a reduced set of nodes with some hierarchy. We need to build self-managing decentralized systems, where data storage needs to be replicated and load balanced across the network in order to provide fault tolerance.

The contribution we present in this paper is Beernet, a development framework for decentralized systems that uses our structured overlay network topology, called Relaxed-ring (Mejías & Van Roy, 2008). The relaxed-ring deals with non-transitive connectivity, making it suitable for Internet applications. On top of the relaxed-ring, we implement a replicated storage with the symmetric replication strategy designed by (Ghodsi, 2006). To keep replicas consistent, we developed a transactional support which offers three different protocols: two-phase commit, paxos consensus algorithm (Moser & Haridi, 2007) and paxos with eager locking. The validation of the first two protocols, and the design and implementation of the last one is also part of our contribution. We will explain in detail why we needed eager locking. We also describe decentralized applications developed with Beernet as validation of the programming framework, and to analyze the different scenarios that need different transactional support.

This work is supported by projects SELFMAN (contract number: 034084) and MANCOOSI (contract number: 214898)

Self Management

The complexity of almost any system is proportional to its size. This rule also holds for distributed systems. As systems grow larger, they become more and more difficult to manage. Therefore, increasing systems' self manageability appears as a natural way of dealing with high level complexity. By self management, we mean the ability of a system to modify itself to handle changes in its internal state or its environment without human intervention but according to high-level management policies. This means that human intervention is lifted up to the level where policies are defined.

Typical self-management operations are: tune performance, reconfigure, replicate data, detect failures and recover from them, detect intrusion and attacks, add or remove parts of the system, which can be components within a process, or a whole peer, and others. Each of those actions or a combination of them can be identified as *self-configuration, self-organization, self-healing, self-tuning, self-protection* and *self-optimization*, often called in literature *self-** properties.

One of the key operations that a system must perform to achieve self-managing behaviour is to monitor itself and its environment. Once relevant information is collected, the system can take decisions over which action to trigger to achieve its goal. Once the action is triggered, the system needs to monitor again to observe the effect of its action, developing a constant feedback loop.

A basic example of a self-managing system working with a feedback loop is air conditioning. A thermometer is constantly monitoring the temperature of the room. The information is given to a thermostat, where the desired temperature has been set. The thermometer does not need to know about the goal temperature, it simple needs to monitor the temperature of the room, so the system is very modularized. When the temperature goes over a threshold value, the thermostat decides to run the cooling down mechanism, which is the actuator of the system. Room's temperature is monitored again to measure the cooling down effect, building a feedback loop.

In peer-to-peer systems, monitoring is distributed and based only on the local knowledge that every peer has. Peers monitor each other and trigger actions in other peers. Since there is no central point of control that observes the whole system at once, global state can be inferred but always as an approximation. Self-managing behaviour must be observed as a property of the whole network, and not as an isolated property of a single peer.

Structured Overlay Networks

A computer network, is a group of interconnected processes able to route messages between them. Internet is a group of interconnected networks, routing messages between processes independently of the network where they belong. An *overlay network* is a network built on top of another network or set of networks. For instance, a group of processes using the Internet to route their message is said to be an overlay network, where the Internet is the *underlay network*. Actually, the Internet itself can be seen as an overlay network running on top of the group of local area networks.

Examples of *unstructured* overlay networks can be found in what it is called the second generation of peer-to-peer networks. This generation is mainly represented by Gnutella (Gnutella, 2003) and Freenet (FreeNet Community, 2003), and it was developed having file-sharing as the main goal. These networks do not rely on any server, and they are able to route messages through their peers independently of the underlay network. This generation was actually a solution to Napster shut down, because there was no server to stop. These systems are known as *unstructured* overlay networks because peers are randomly connected without any particularly defined structure. As we have discussed already, nowadays almost any machine can behave as a client and a server. Therefore, every peer can trigger queries as a client, and handle queries from other peers, playing the role of a server.

The algorithm to route messages in such unstructured network is called *flooding*. It is very simple but highly bandwidth consuming. It works as follows: the peer that triggers the query sends it to all its neighbours with a time to live (TTL) value. The TTL can be expressed in seconds or hops. A hop is what it takes for a message to go from one peer to another that is directly connected in the overlay (we do not include the amount of hops involved in the underlay network using TCP/IP connection). The growth in the amount of messages is exponential, and many peers have to process the same query several times, making the algorithm very inefficient. Another known problem is related to non-popular items that are difficult to find if the TTL is not large enough. Unfortunately, the larger the TTL, the more congested the network.

Flooding routing works fine for networks with a tree topology, because it avoids that peers receives messages more than once, but it is very costly for unstructured overlay networks, being inefficient in bandwidth and processing-power usage. Another problem is that there is no guarantee of reachability or consistency properties, which we consider important to build decentralized systems that constantly update the values of the stored items. Even though Gnutella can keep large amount of peers connected, it does not mean that scalable services can be built on top of it because of the problems on efficiency, reachability and consistency (Markatos, 2002; Ripeanu, Foster, & Iamnitchi, 2002). Also, according to (Daswani & Garcia-Molina, 2002), it is not difficult to perform a query-flood DoS attack in Gnutella-like networks, but their success depend on the topology of the network and the place of the originator of the attack, which is related to the reachability issue we already discussed.

Distributed Hash Tables

The third generation of peer-to-peer networks, also known as *structured overlay networks* (SONs), is the result of academia's interest in peer-to-peer systems. It clearly aims to solve the problems of unstructured networks by providing efficient routing, guaranteeing reachability and consistent retrieval of information. Adding structure makes it possible to achieve these improvements, but it also creates new challenges such as dealing with disconnections of peers and non-transitive links. SONs typically provide a distributed hash table (DHT) where every peer is responsible for a part of the hash table. There are two basic operations that every DHT must provide: `put(key, value)` and `get(key)`. The `put` operation stores the value associated with its key such that every peer can retrieve it with the `get` operator. If another value was already stored under the same key, the value is overwritten.

Chord (Stoica, Morris, Karger, Kaashoek, & Balakrishnan, 2001; Dabek et al., 2001) is one of the most known and referenced SON. By describing and analyzing Chord in this section, we will also review the main principles behind many others ring-based peer-to-peer systems, such as Chord# (Schütt, Schintke, & Reinefeld, 2007), DKS (Ghodsi, 2006), OpenDHT (Rhea et al., 2005), P2PS (Mesaros, Carton, & Van Roy, 2005) and the Relaxed-Ring (Mejías & Van Roy, 2008). In Chord, peers are self-organized forming a ring with a circular address space of size N. Hash keys are integers from 0 to $N - 1$. The ring can be seen as a double-linked list with every peer having two basic pointers: *predecessor* and *successor* (abbreviated as *pred* and *succ*). Figure 1 depicts an example of a Chord ring. Only pointers of peer identified with id q are drawn on the figure but every peer holds equivalent pointers. Peers p and s corresponds to *pred* and *succ* respectively. This means that $p < q < s$, where '<' is defined on the circular address space following the ring clockwise.

The ring provides a DHT where every peer is responsible for the storage of a set of keys determined by its own id and its predecessor. In the case of q, the peer is responsible for the range $(p, q]$, (i.e., excluding pred's id and including its own). If the ring is perfectly linked, there is no overlapping of peers' responsibilities, and therefore, every lookup operation gives consistent results.

To provide efficient routing, Chord uses a set of extra pointers called *fingers* or *long references*. They are chosen dividing the address space in halves. The farther finger of q is the responsible of key $(q + N/2)$ *mod* N. In our example in Figure 1, we consider $q = 0$ to label finger keys, and therefore, the ideal farther finger key is $N/2$. In the figure there is no peer holding exactly that key, but peer k is currently the responsible. Closer fingers are chosen using the same formula but dividing N by powers of 2. The fingers, together with pointers to pred and succ, form the routing table of a peer. Ideally, every peer holds references to $log_2 N$ fingers.

Figure 1 shows three different events producing *churn*. Churn means that new peers constantly join the ring, and some peers also leave the ring. The events we observe are: peer j joins as k's predecessor, peer b leaves voluntarily the network, and peer m crashes. In the case of the join, k accepts j only if j belongs to k's range of responsibility. Since $N/2 > j > q$, peer j becomes the new responsible of $N/2$, and therefore, it is a more suitable finger for q. This value needs to be updated somehow. A similar situation occurs when b leaves, because c becomes the new responsible of $N/4$. The difference here is that now q has temporary no finger for that value until it knows about c. The crash of m does not affect q's routing table, but it surely affect other peers' routing table, and the responsibility of m's successor. These events help us to understand the complexity of the system. We will discuss now the existing strategies to deal with these events.

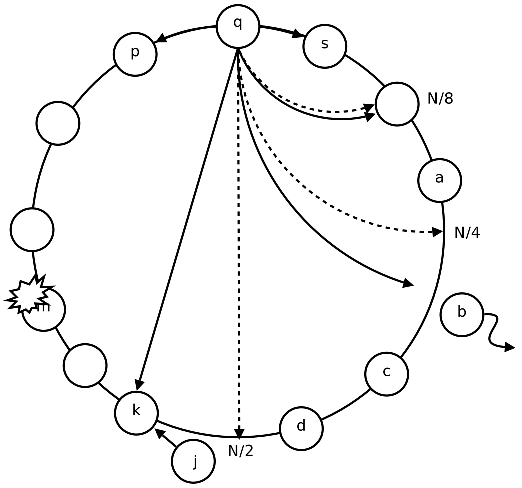

Figure 1. Example of Chord ring with some events causing churn.

Self Organization and Self Healing

The ring maintenance is addressed differently by each SON. We know that Chord, OpenDHT and Chord# rely on periodic stabilization to fix successor, predecessor and finger pointers. Periodic stabilization constantly checks the predecessor pointer of the successor of each peer. If a new node is detected in between two nodes, the predecessor notifies its new successor fixing the ring. Fingers are also queried about its predecessor. If the predecessor is closest to the ideal finger key, the routing table is updated. Such strategy has the advantage of treating leaves as failures. Therefore, there is no need to define a protocol for gentle leaves, because pointers will be fixed in the next round of stabilization. However, this implies that stabilization needs to be run often enough, increasing bandwidth consumption. With respect to routing, it has been shown by (Krishnamurthy & Ardelius, 2008) that logarithmic routing cannot be guaranteed if the period for stabilization is larger than a certain threshold. More problematically, it has also been shown by (Ghodsi, 2006) that lookup inconsistencies can appear in Chord just because of churn, even if failures do not occur. This is a serious problem for correctness. To avoid this problem, DKS introduces the concept of correction-on-change, meaning that pointers are fixed as soon as a failure, leave or join is detected. Peers do not wait for a periodic check. To avoid lookup inconsistencies, DKS defines a protocol for atomic join/leave, requiring that a peer respect the locking protocol before it leaves the network. Unfortunately, this strategy is not very fault-tolerant, and it relies on peers not leaving the network before they acquire the needed locks.

The relaxed-ring also follows the correction-on-change strategy for ring maintenance, but it does not rely on locks, and more importantly, it does not rely on transitive connectivity. The strategy is to allow branches when a peer is not able to connect to its predecessor. The ring does not need to be perfectly double link to work, but every peer needs to know who its successor is in order to provide lookup consistency. The relaxation of the ring implies a degradation on the routing complexity, becoming $log(N) + b$, where b is the size of the branch where the searched node is locate. Empirical evaluation (Mejías & Van Roy, 2008) estimates that $b < 1$, so the degradation is very small.

Replication strategies

Plain DHT is not enough to provide a fault tolerant storage service. Some sort of replication is needed to keep every item in the network even when a responsible leaves the system. Some of the used strategies are *successor list*, *leaf-set*, *multiple hashing* and *symmetric replication*. The most basic mechanism is probably the first one proposed in (Stoica et al., 2001), where $log_2 N$ replicas are stored on the successor list of the responsible of each key. When a node fails, the successor takes over the responsibility, and therefore, it is a good idea that the successor stores the replicas of the items. A very similar strategy is the one used by networks having an overlay topology like Pastry, using the leaf-set (Rowstron & Druschel, 2001a, 2001b) for storing the replicas. The leaf-set is composed by $log_2 N/2$ successor and $log_2 N/2$ predecessors. This strategy also generates a different replica set for each peer. It has the same advantages as in the successor list strategy, because it does not add extra connections, and the peer that should take over the responsibility in case of a failure already has the values of the replicas.

There are two main disadvantages on these two schemes. First of all, churn introduces many changes on the participants of the replica sets. Each join/leave/fail event introduces changes in $log_2 N$ replica sets, affecting peers that are not directly involved with the churn events. The second disadvantage is that there is a unique entry point for each replica set. To find the successor list of the responsible of a key, first you need to find who is the responsible. This means that the main peer of the replica set is a point of congestion.

CAN (Ratnasamy, Francis, Handley, Karp, & Schenker, 2001) and Tapestry (Zhao et al., 2003) used multiple hashing as replication strategy. The idea is that every item is stored with different hash functions known to all peers in the network. One disadvantage claimed in (Ghodsi, 2006) is that you need to know the inverse of the hash functions to recover from failures. A more crucial disadvantage is the lack of relationship between the replica sets per item. There will be a different replica set for almost every item stored in the network, making the reorganization of replicas under every churn event very costly.

Symmetric replication is a simple and effective replication strategy presented in (Ghodsi, 2006) with several advantages and few disadvantages. First of all, it does not have an entry point of congestion as with the successor list and the leaf sets. Members of the replica set are not indirectly affected by churn, and as in multiple hashing, the replicas are spread across the network, with the advantage that they are symmetrically placed using a regular polygon of f sides, where f is the chosen replication factor. This strategy provides an easier way of finding the replicas, and it balances the load more uniformly. A disadvantage shared by multiple hashing and symmetric replication is that both rely on a uniform distribution of peers on the address space. However, this assumption is very reasonable since many SONs also rely on this property in order to achieve the promised logarithmic routing.

Beernet

Beernet stands for pbeer-to-pbeer network, where words peer and beer are mixed to emphasise the fact that this is a peer-to-peer network built on top of a *relaxed*-ring topology, considering that beers are usually a mean to achieve relaxation. Beernet is globally organized as a set of layers providing higher level abstract with a bottom-up approach. Figure 2 gives the global picture of how components are organized. Components are objects running on their own lightweight thread of execution and they all communicate asynchronously through message passing. They are basically actors, as in the actor model of (Hewitt, Bishop, & Steiger, 1973).

The bottom layer is the *Network* component. This actor is composed by four other actors. The most basic communication is provided by perfect point-to-point link (*Pp2p link*) that simply connects two ports. The *Peer-to-peer link* allows a simpler way of sending messages to a peer using its global representation, instead of extracting the port explicitly every time a message is to be sent. *Peer-to-peer link* uses *Pp2p link*. Network uses two failure detectors: one provided by Mozart, and the other one implemented in Beernet itself. The *Mozart failure detector* takes advantage of the fault-stream (Collet & Van Roy, 2006) of every distributed entity. *Beernet failure detector* is built as a self-tuning failure detector that uses its own protocol to change the frequency and timeout values of the keep alive messages. Both failure detectors are eventually perfect, meaning that they are strongly complete, and eventually accurate.

The Relaxed-Ring component uses the Network component to exchange messages between directly connected peers, and to detect their failures. It has two main components: the *Relaxed-Ring maintenance* and the *Finger Table*. The relaxed-ring maintenance could actually be replace by a component implementing any of the network topologies we described in Section . We use the relaxed-ring because it is cost efficient and it does not rely on transitive connectivity. The finger table component is in charge of efficiently routing messages that are not sent neither to the successor nor the predecessor of a node.

The reliable message-sending layer is implemented on top of the relaxed-ring maintenance. This layer includes the basic services *Reliable Send*, *Multicast* and *Broadcast*. Each of them is running on its own actor, but they can collaborate if

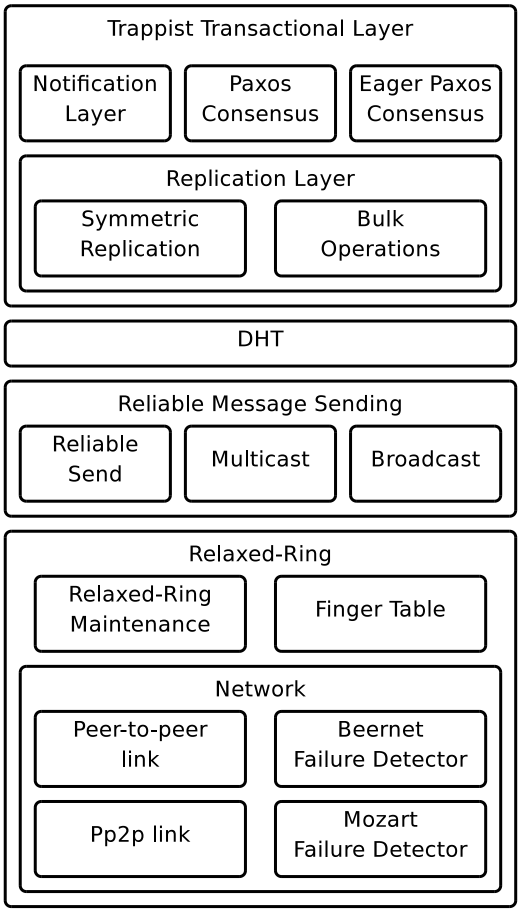

Figure 2. Beernet's actor architecture. Every component run on its own lightweight thread and they all communicate asynchronously through message passing.

necessary, as the relaxed-ring maintenance collaborate with the finger table. The basic *DHT* with its *put* and *get* operations is implemented on top of the messaging services.

In Beernet, we have decided to implement the transactional layer, Trappist, having the replication layer as part of the Trappist component. This is a major difference with Scalaris (Schütt, Schintke, & Reinefeld, 2008), because they present their architecture having replication and transaction as two independent layers. We claim that replication needs the transactional mechanism in order to restore replicas in case of failures. Having the knowledge of the protocol that is used to manage the replica is the best option to keep replica maintenance efficient. That is why in Beernet the replica maintenance also belongs to the transactional layer instead of being an independent component.

There are still some orthogonal components within the replica management that can be changed by equivalent ones. For instance, we have chosen to work with *Symmetric replication* instead of successor list replication, or leaf set replication. To reach the replicas, the transactional layer will use the *Bulk operations* (Ghodsi, 2006) which can be written for any replication strategy.

The Trappist layer includes three different protocols to provide transactional support: *Paxos Consensus*, *Eager Paxos Consensus* and *Two-Phase Commit*. The three of them are explained in detail in the next section. Two-phase commit is not included in Figure 2 because its use is not recommended for building applications. We have implemented it for purely academic purposes. These protocols are enriched by a *Notification Layer* component that contributes to develop more synchronous collaborative applications.

Transactional Replicated Storage

The most basic operations provided by a DHT are `put(key value)` and `get(key)`. We have seen in Section that this is not enough to provide fault tolerance, and that a replication strategy should be used in order to guarantee data storage. Replicas are not simple to maintain independently of the chosen replication strategy. Therefore, it is very convenient to add transactional support to the DHT so as to manage the replicas, and to provide atomic operations over a set of items.

The two-phase commit protocol (2PC) is one of the most popular choices for implementing distributed transactions, being used since the 1980s. Unfortunately, its use on peer-to-peer networks is very inefficient because it relies on the survival of the transaction manager, as we will explain further in this section. A three-phase commit protocol (3PC) has been designed in order to overcome the limitation of 2PC. However, 3PC introduces an extra round-trip which results in higher latency and increased message load. We will see how transactional support based on Paxos consensus (Moser & Haridi, 2007; Gray & Lamport, 2006) works well in decentralized systems. This algorithm is especially adapted for the requirements of a DHT and can survive a crash of the coordinator during a transaction. Compared to 3PC, it reduces latency and overall message load by requiring less message round-trips.

We extends the Paxos consensus algorithm with an eager locking mechanism, so as to fit the requirements we identify in synchronous collaborative applications. A notification layer is also added to the transactional layer support, which can be used by any of the transactional protocols we will describe.

Two-Phase Commit

The pseudo-code in Algorithm 1 implements a swap operation within a transaction. The objective is that the instructions from the beginning of the transaction (`BOT`) until its end (`EOT`) are executed atomically to avoid race conditions with other concurrent operations. The values of *item_i* and *item_j* are stored on different peers. The operators *put* and *get* are replaced by *read* and *write* in order to differentiate a regular DHT from a transactional DHT. Since the operations have different semantics, it is justified to use different keywords.

In order to guarantee atomic commit of a transaction on a decentralized storage, two-phase commit uses a *validation* phase and a *write* phase, coordinated by a *transaction manager* (TM). All peers responsible for the items involved in

Algorithm 1 Swap transaction

```
BOT
    x = read(item_i);
    y = read(item_j);
    write(item_j, x);
    write(item_i, y);
EOT
```

the transaction, as well as their replicas, become *transaction participants* (TP). Initially, the TM sends a request to every TP to *prepare* the transaction. If the item is available, the TP will lock it and acknowledge the *prepare* request. Otherwise, it will reply *abort*. The *write* phase follows *validation* once the replies are collected by the TM. If none of the participants voted *abort*, then the decision will be *commit*. When the participants receive the commit message from the TM, they will make the update permanent and release the lock on the item. An abort message will discard any update and release the item locks.

The problem with the 2PC protocol is that relies too much on the survival of the transaction manager. If the TM fails during the validation phase, it will block all the TPs that acknowledged the prepare message. A very reliable TM is required for this protocol, but it cannot be guaranteed on peer-to-peer networks. Figures 3 and 4 depict 2PC protocol showing two possible executions. The diagrams do not include the client, but they concentrate on the interaction between the TM and the TPs. Figure 3 shows a successful execution of the protocol where the TPs get the confirmation of the TM about the result of the transaction. Figure 4 spots the main problem of this protocol. If the TM crashes after collecting the locks of the TPs, the TPs remained locked forever if the algorithm is *crash-stop*. PostgreSQL (PostgreSQL Global Development Group, 2009), a well established object-relational database management system, implements 2PC as a *crash-recovery* algorithm, meaning that the TM can reboot and recover the state before the crash to continue with the protocol. Discussing with PostgreSQL developers, we have learned that a transaction could hang for a whole weekend before the locks are released again. This kind of behaviour is not feasible in peer-to-peer networks when there is no certainty that a peer that leaves the network will ever come back.

Paxos Consensus Algorithm

The 3PC protocol avoids the blocking problem of 2PC at the cost of an extra message round-trip. This solution might be acceptable for cluster-based applications but not for peer-to-peer networks, where it is better to have less rounds with more messages than adding extra rounds to the protocol. This problem led to the recent introduction of a protocol for atomic transactional DHT, by (Moser & Haridi, 2007), based on Paxos consensus (Gray & Lamport, 2006).

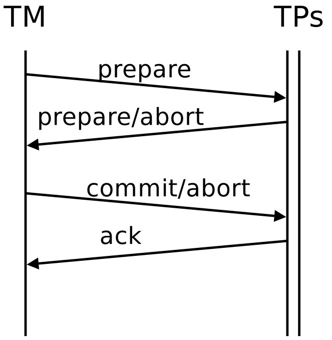

Figure 3. Two Phase Commit protocol reaching termination

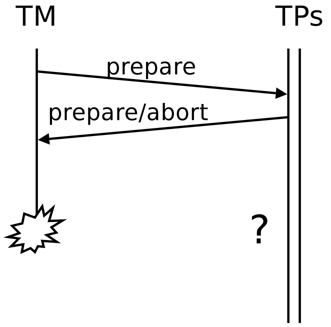

Figure 4. Two Phase Commit protocol not knowing how to continue or unlock the replicas because of the failure of the transaction manager.

The idea is to add replicated transaction managers (rTM) that can take over the responsibility of the TM in case of failure. The other advantage is that decisions can be made considering a majority of the participants reaching consensus, and therefore, not all participants need to be alive or reachable to commit the transaction. This means that as long as the majority of participants survives, the algorithm terminates even in presence of failures of the TM and TPs, without blocking the involved items.

Figure 5 describes how the Paxos-consensus protocol works. The client, which is connected to a peer that is part of the network, triggers a transaction in order to read/write some items from the global store. When the transaction begins, the peer becomes the transaction manager (TM) for that particular transaction. The whole transaction is divided in two phases: *read phase* and *commit phase*. During the *read phase*, the TM contacts all transaction participants (TPs) for all the items involved in the transaction. TPs are chosen from the peers holding a replica of the items. The modification to the data is done optimistically without requesting any lock yet. Once all the read/write operations are done, and the client decides to commit the transaction, the *commit phase* is started.

In order to commit the changes on the replicas, it is necessary to get the lock of the majority of TPs for all items. But, before requesting the locks, it is necessary to register a set of replicated transaction managers (rTMs) that are able to carry on the transaction in case that the TM crashes. The idea is to avoid locking TPs forever. Once the rTMs are registered, the TM sends a *prepare* message to all participants. This is equivalent to request the lock of the item. The TPs answer

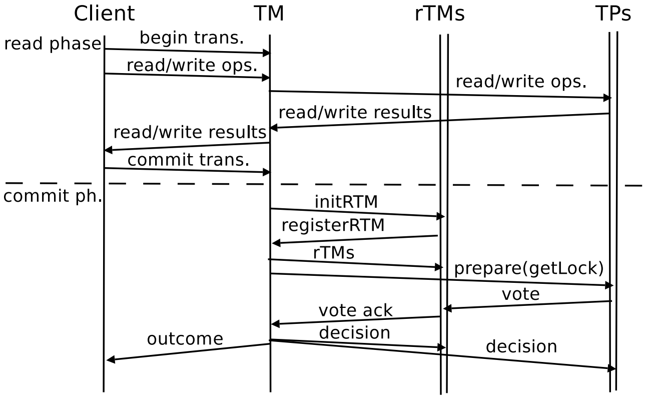

Figure 5. Paxos consensus atomic commit on a DHT.

back with a *vote* to all TMs (arrow to TM removed for legibility). The vote is acknowledged by all rTMs to the leader TM. The TM will be able to take a decision if the majority of rTMs have enough information to take exactly the same decision. If the TM crashes at this point, another rTM can take over the transaction. The decision will be *commit* if the majority of TPs voted for commit. It will be *abort* otherwise. Once the decision is received by the TPs, locks are released.

The protocol provides atomic commit on all replicas with fault tolerance on the transaction manager and the participants. As long as the majority of TMs and TPs survives the process, the transaction will correctly finish. These are very strong properties that will allow the development of collaborative applications on a decentralized system without depending on a server.

Self-management We can observe the property of self-configuration in this transactional protocol in the way the replicated transaction managers are chosen, and in the way the replicas are found. Even when replicas should not change from one transaction to the other, unless there is some churn, the set of TM and rTMs tends to be different in every transaction. There is no intervention in the election of the members of these sets, they just follow the high level policies and self-configure to run the transaction. The self-healing property can be observed when the TM fails. One of the rTM is elected to become the new TM, and it finishes the transaction. The election is done following the identifiers in the ring, so they all reach an agreement.

Paxos with Eager Locking

We have observed how Paxos consensus algorithm for atomic transactions on DHTs is extremely useful for building systems with decentralized storage based on symmetric replication. The protocol works very well for applications such as Wikipedia on Scalaris (Schütt et al., 2008; Plantikow, Reinefeld, & Schintke, 2007) or the recommendation system Sindaca, presented in Section . These systems are designed to support asynchronous collaboration between application's users. The fact that Paxos consensus protocol works with optimistic locking fits well asynchronous collaboration.

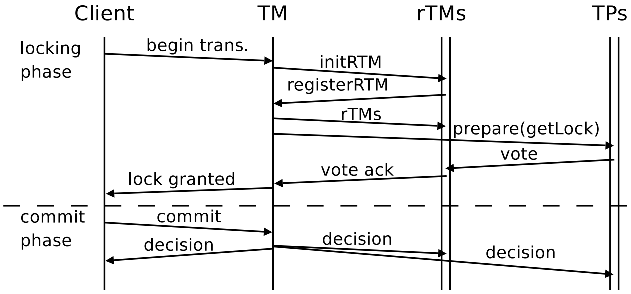

Figure 6. Paxos consensus with eager locking and notification to the readers.

However, this locking strategy limits the functionality of synchronous collaborative applications such as DeTransDraw, a collaborative drawing tool that we will describe more in detail in Section .

DeTransDraw has a shared drawing area where users actively make updates and observe the changes made by other users. If two users make modifications to the same object at the same time, at the end of the their work, when they decide to commit, only one of them will get her changes committed, and the other one will loose everything. Because users are working synchronously, the probability that this happens is much larger than in applications such as Wikipedia. This is why a pessimistic approach with eager locking is needed.

We have adapted Paxos to support eager locking adding a notification mechanism for the registered readers of every shared item. We have implemented this new protocol in Trappist, the transaction layer support of Beernet, with the possibility of dynamically choosing between the two Paxos protocols. Given this choice, the application can decide the protocol to be used depending on the functionality that is provided to the users.

Figure 6 depicts the adapted protocol with eager locking. The read-phase and commit-phase from the original protocol has been replaced by *locking-phase* and *commit-phase*. The read phase disappears because the transaction manager tries eagerly to get the relevant locks to proceed with the transaction. Once the locks are collected, the client is informed of the result. The goal is to prevent users from trying to start working on items that are already locked. The client of the transaction starts working on the changes on the items as soon as the transaction begins. Starting to work on an item is actually the trigger of the transaction.

When the user stops making modifications, it triggers the commit-phase. The transaction manager can take the decision immediately because the majority of the votes have been already collected at this stage. The decision is propagated to the client, the replicated transaction managers and transaction participants, as in the original Paxos algorithm. As there is no read-phase, it is important that the decision is transmitted to the TPs and rTMs together with the new state of the item, and not only a *commit/abort* message.

This protocol is unfortunately more fragile than Paxos without eager locking. By dividing the acquisition of locks and the decision of commit, we can propagate information to

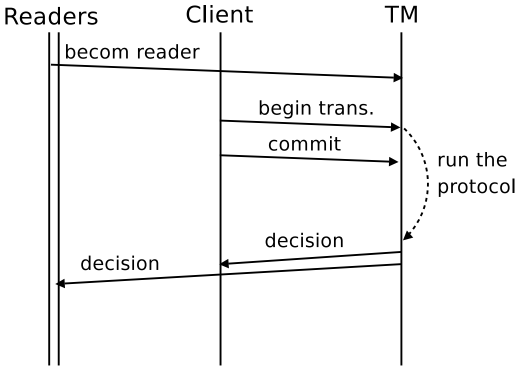

Figure 7. Notification layer protocol. Peers register to each item by *becoming readers*. The figure shows one notification for the decision if the transaction is run with Paxos.

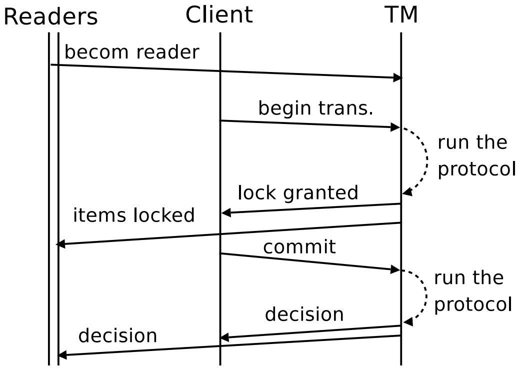

Figure 8. Notification layer protocol. The figure shows notifications for locking and decision, if transaction is run with Eager Paxos.

the readers more eagerly, but we increase the period of time where the TM and the client need to survive. If the client fails before committing its final value, the locks need to be retrieved and released. If there is a false suspicion, we could end up having two clients claiming the lock of an item. To solve this, it is necessary to add timestamps not only to the values of the items, but also to the locks. If the TM crashes before the client commits its final decision, one of the rTM becomes the new TM, and it needs to inform the client about the failure recovery. Once the client is notified about the recovery, the client can commit the transaction using the new TM.

Self-management The self-configuration property with respect to the set of rTMs is inherited from the previous Paxos protocol without eager locking. Self-healing is also achieved, because the system can recover from the crash of the client and the TM, and it can complete if the majority of participants survives. The mechanism is a bit more complex due to the split of the locking and commit phase.

Notification Layer

The modification with eager locking provides notification to the readers every time an item is locked and updated. Sometimes it is not necessary to get a notification on locking, and only the update is important. In such case, it is interesting to have a layer of notification independent of the protocol used to update the item. This kind of feature is useful to implement applications such an online score board, where only a few peers modify the state of the application, and many peers participate as readers. For the readers is not necessary to get a notification that some value is currently being update. They just need to get the last value of the item.

The layer consists of a reliable multicast that sends a notification to all subscribed readers of an item. In order to make the multicast efficient, if the amount of readers is smaller than $log(N)$, a direct message sending can be performed. If the amount of readers is larger, the update message can be transmitted using the multicast layer of the peer-to-peer network.

Trappist

As we have previously mentioned in this section, the transactional layer implementing these three protocols is called Trappist, which stands for Transactions over peer-to-peer with isolation, where isolation means that transactions are atomic and with concurrency control. In this section we show how to use the transactional support of Beernet, which is implemented with the Mozart (Mozart Consortium, 2008) programming system. By describing Trappist's API, we also analyse the high level abstractions provided by the system, and how replica maintenance is hidden from the programmer. We start by creating a Beernet peer. Currently, nodes in Beernet are created by default with transactional support. However, to prevent conflicts with previous versions we explicitly flag transactions to be included in the following example:

```
functor
import
   Pbeer at 'Pbeer.ozf'
define
   Node = {Pbeer.new args(transactions:true)}
   ...
```

The most basic support provided by Beernet corresponds to the DHT operations *put* and *get*. This operations do not replicated the value of the item, but they are also part of the implementation of the transactional layer which actually realizes the replication. What follows is an example of how put and get can be used.

```
{Node put(key value)}
Value = {Node get(key $)}
```

To use the transactional layer, the user must write a procedure with one argument, typically named TM. This argument represents a transactional object, which is an instance of the transaction manager that triggers the transaction. The object receives the operations *read* and *write*, which are almost equivalent to *put* and *get*. The main semantic difference between the operations is that if the transaction is aborted, *write* has no effect on the stored data. And if the transaction succeeds, the value is written at least on the majority of the replicas. Other operations received by the transactional object are *commit* and *abort*, to explicitly trigger those actions on the protocol. The operation *remove* is also implemented in order to delete an item from the DHT.

To run the transaction, the user must invoke the method *executeTransaction*, which receives three arguments: the procedure containing the operations, a port to receive the outcome of the transaction, and the protocol to be used for running the transaction. Note that at the creation of the node, we did not specify the protocol to be used by every transaction. This is because the protocol can be chosen dynamically, allowing the users to choose the best suitable protocol for every functionality. We consider this to be a very important property of our system. With respect to the procedure containing the transactional operations, it is the equivalent to the pseudo code we presented in Algorithm 1. One difference is that the commit operation needs to be explicitly triggered, being the equivalent to `EOT` (end of transaction). The advantage of making the commit explicit, is that transactions that only read values do not need to run the commit phase.

Algorithm 2 is a complete example for writing two items with key/value pairs: *hello*/*"Charlotte"* and *foo*/*bar*. The outcome of the transaction appears on variable *Stream*, which is the output of port *Client*. If the outcome of the transaction is `commit`, it guarantees that both items where successfully stored at least in the majority of the correspondent replicas.

Algorithm 2 Using transactions with Paxos consensus to write two items

```
declare
Stream Client
Trans = proc {$ TM}
               {TM write(hello "Charlotte")}
               {TM write(foo bar)}
               {TM commit}
          end
{NewPort Stream Client}
{Node executeTransaction(Trans Client paxos)}
if Stream.1 == commit then
     {Browse "transaction␣succeeded"}
end
```

To retrieve the values the user passes a variable which has no value yet. The value is bound by the transactional object. Algorithm 3 shows how to retrieve the values stored under keys *hello* and *foo*.

Note that it is not necessary to catch exceptions using Beernet, because the outcome is reported on the stream of the client's port. If there is a failure on the transaction, the outcome will be `abort`, and the user will be able to take the corresponding failure recovery action. If the item is not found, the variable used to retrieve the value is bound to a *failed value*. This language abstraction will raise an exception whenever is used. Like this, exceptions are triggered in the calling site, and not at any of the peers. Now, to prevent catching exceptions when using the value, the Mozart programming system provides Boolean checkers to test whether a variable is bound to a failed value or not.

Algorithm 3 Using transactions with Paxos consensus to read two items

```
declare
V1 V2
Trans2 = proc {$ TM}
                {TM read(hello V1)}
                {TM read(foo V2)}
           end
{Node executeTransaction(Trans2 Client paxos)}
{Browse "for␣hello␣I␣got"#V1}
{Browse "for␣foo␣I␣got"#V2}
```

Applications

This section describes applications designed and implemented using Beernet: Sindaca, DeTransDraw and a small decentralized wiki. The first one, Sindaca, uses intensively the transactional DHT layer Trappist. DeTransDraw benefits from the Eager locking protocol in order to provide synchronized collaboration. The small wiki was designed and implemented by students during a course in distributed programming. We present them here to show the impact of the contribution of this work.

Sindaca

This section presents the design and functionality of our community-driven recommendation system named Sindaca, which stands for Sharing Idols N Discussing About Common Addictions. The name spots the main functionality of this application which is making recommendations on music, videos, text and other cultural expressions. It is not designed for file sharing to avoid legal issues with copyright. It allows users to provide links to official sources of titles. Users get notifications about new suggestions, and they can vote on the suggestions to express their preferences.

We have implemented a web interface to have access to Sindaca. All requests done through the web interface are transmitted to a peer in the network which triggers the corresponding operations in the peer-to-peer network. The results are transmitted back to the web server, which presents the information in HTML format as in any web page. Using a web interface to transmit information between the end user and the peer-to-peer network has been used previously in various projects. A very related one is the peer-to-peer version of the Wikipedia, implemented using Scalaris (Plantikow et al., 2007; Schütt et al., 2008). We have extended this architecture with a notification layer which allows eager information updates. This layer is also used in the DeTransDraw application, as we will see in Section . However, this eager notification feature is not provided on the web interface.

To generalize the similitudes and differences between Sindaca and the above mentioned applications, we can say the following: the Wikipedia on Scalaris uses *optimistic* transactions using the Paxos consensus algorithm. DeTransDraw uses *pessimistic eager-locking* transactions using Paxos consensus algorithm with a *notification layer*. Sindaca is a combination of those strategies. It uses *optimistic* transactions with Paxos algorithm extended with the *notification layer*, both implemented in Trappist.

It is important to remark that Sindaca is not implemented on top of a database supporting SQL queries. Sindaca is implemented on top of a transactional distributed hash table with symmetrically replicated state. Therefore, the basic unit for storage is the key-value pair, which is what it is called *item*. The information of every user is stored as one item. The *value* of such item is a record with the basic information: user's id, username and password. We have chosen a very minimal record to build the prototype, but the value can potentially store any data such as user's real name, contact information, age, description, etc. The *key* of the item is an *Oz name* (Mozart Consortium, 2008), which is unique and unforgeable, acting like a capability reference, as in (Miller & Shapiro, 2003). This strategy provides us certain level of security, because only programs that are able to map usernames with their capability can have access to the key, and therefore, access to the item. The username-capability mapping is only available to programs holding the corresponding capability to the mapping table.

Creating a user Code 4 shows the transaction to create a user. First of all, it is necessary to read the list of users to verify that the new username is not already in use. This is done by reading the item under key `users`, and verifying if `Username` is a member of it. In such case, the transaction is aborted with the operation `{TM abort}`. If the transaction continues, we read the item `nextUser` to get a user identifier. Then, we create a new item with the *capability key* `UserCap`. Afterwards, the value of the `nextUser` item is incremented, and the item with the list of users is also updated.

This code helps us to describe the expressiveness of the transactional support, and how the issues concerning replication, configuration and failure handling, are hidden from the programmer.

Jalisco transactions Apart from the code to create users, there are other transactions in charge of adding recommendations done by user, or adding votes from a user about others recommendation. The outcome of a transaction is either *abort* or *commit*. This outcome will be sent to a port where the application will decide the next step. When the transaction to create new users aborts because the username is already in use, the application will need to request the new user to choose a different username before attempting to run a new transaction. In the case of creating new recommendation and voting, getting abort as outcome of the transaction only means that there where some concurrent transactions that committed first, creating a temporary conflict with our transaction. In such case, the transaction can be retried without any modification until it is committed. To simplify the process of retrying, we have implemented the procedure *Jalisco*, which comes from the Mexican expression "Jalisco nunca pierde" (*Jalisco never loses*). This procedure will simply retry a transaction until it is committed. The code is shown in Code 5.

Algorithm 4 Creating a new user.

```
proc {CreateUser TM}
      Users UserId UserCap
   in
      UserCap = {NewName}
      {TM read(users Users)}
      if {IsMember Username Users} then
         UsernameInUse = true
         {TM abort}
      else
         {TM read(nextUser UserId)}
         {TM write(UserCap user(username:Username
                                 id:UserId
                                 passwd:Passwd
                                 cap:UserCap
                                 recommed:nil
                                 votes:nil
                                 voted:nil))}
         {TM write(nextUser UserId+1)}
         {TM write(users {AddTo Users
                                Username UserCap})}
         {TM commit}
      end
   end
```

Algorithm 5 Jalisco transaction retries a transaction until it is committed

```
fun {Jalisco Trans}
   P S
   proc {InsistingLoop S}
      {ThePbeer executeTransaction(Trans P paxos)}
      case S
      of abort|T then
         {InsistingLoop T}
      [] commit|_ then
         commit
      end
   end
in
   {NewPort S P}
   {InsistingLoop S}
end
```

The function creates a port to receive the outcome of the transaction. The `InsistingLoop` executes the transaction on the peer `ThePbeer`, and it waits on the stream of the port to check the outcome of the transaction. If it is abort, it just continues with the loop. If it is commit, it simply returns that the transaction has committed.

This is simply a design pattern to be used in transactional

DHTs. It can be seen as a very simply feedback loop. The outcome of the transaction is what is being monitored. The action to be taken in case of `abort` is to insist on running the transaction until the relevant locks are granted. Once they are granted and the message `commit` is monitored, the feedback loop ceases to monitor.

DeTransDraw

DeTransDraw is a decentralized collaborative graphical editor with a shared drawing area. It provides synchronous collaboration between users with graphical support for notifications about other users' activities. Conflict resolution is achieved with a decentralized transactional service with storage replication, and self-management replication for fault-tolerance. The transactional service also allows the application to prevent performance degradation due to network latency, which is an important feature for synchronous collaboration.

DeTransDraw is implemented on top of Beernet, and it uses the paxos consensus algorithm with eager locking. Since Beernet provides a DHT, the drawing information has to be stored in form of items. Each drawing object is an item where its identifier is the key, and the value corresponds to the position, shape, colour, and other properties of the figure. The application has been implemented in our research group, being Jérémie Melchior the main developer.

Figure 9 shows the protocol of the application, where the client talks to the peer-to-peer network through the transaction manager. The protocol is an instance of Eager Paxos consensus algorithm, as it is described in the previous section, combined with the notification layer that communicates with the readers. In this case, the readers are all the other users of DeTransDraw. We can observe that the `client` tries to acquire the locks of some figure with `begin transaction`. Confirmation of acquiring the locks is `locked granted`. The `commit` message triggers the update of the figures with the modifications done by the user. As we mentioned already, the TM is different for every transaction, and the set of replicated TMs is chosen with the same strategy as symmetric replication. The key to generate the replica set is the one of the TM. The transaction participants (TPs) are all the peers storing a replica of the drawing objects involved in the transaction. Therefore, two concurrent transactions modifying disjoint sets of drawing objects could have completely different sets of TM, rTMs and TPs.

Figure 10 shows how the action of selecting drawing objects changes the state of the network. The figure shows four application windows. The window at the top left corner is a screenshot of PEPINO (Grolaux, Mejías, & Van Roy, 2007), an application that monitors the network and shows it state. In this case, the network is composed by 17 peers. The other three windows are instances of DeTransDraw which are connected to the network. Looking at the tool bars, we can deduce that the user at the top right corner draw the light-gray oval, the user at the bottom left draw the small dark square, and the highlighted user at the bottom right corner draw the large gray rectangle. This highlighted user has selected the two ovals acquiring the correspondent locks. We observe in PEPINO some dark peers, and some other in slightly gray. The peers in blue are the transaction participants which are currently locked. They are the replicas storing the state of the two ovals. Peers in cyan are the replicated transaction manager, being the peer in green the transaction manager for this operation. The other users do not see the modification of the position of the figures, because the other user has not committed yet its modification. Once the modifications are committed, the locks are released, and the new state of the ovals is replicated.

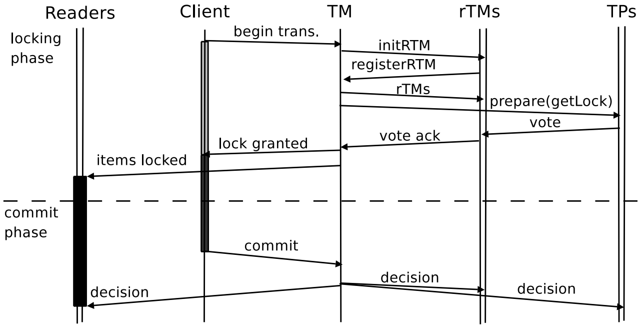

Figure 9. DeTransDraw coordination protocol. It combines optimistic and pessimistic approach, using Trappist's eager locking Paxos and the notification layer to propagate the information to the registered readers.

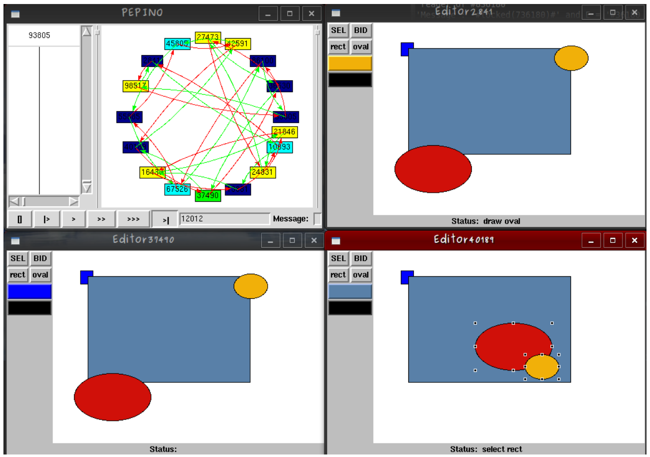

Figure 10. Locking phase in Detransdraw. The user with highlighted window has selected two figures to move them on the drawing. Dark peers on the ring show where the locked replicas are.

The software still needs more development to become a real drawing tool, but it is well advanced as a proof of concepts concerning its decentralized behaviour. It minimizes the impact of network latency, allowing collaborative work with conflict resolution achieved with transactional protocols. It does not have any single point of congestion or failure, because every transaction has its own transaction manager, with a set of replicated transaction managers symmetrically distributed through the network. State is also decentralized on the DHT, having each item symmetrically replicated. Each transaction guarantees atomic updates of the majority of the replicas.

Decentralized Wikipedia

Wikipedia (Wikimedia Foundation, 2009) is an online encyclopedia written collaboratively by volunteers, reaching currently more than 13 million articles. A large community of users constantly updates the articles and create new ones. Such system can certainly benefit from scalable storage and atomic commit, being a good case study for self-organizing peer-to-peer networks with transactional DHT. A fully decentralized Wikipedia (Plantikow et al., 2007) was successfully built with Scalaris (Schütt et al., 2008), which is based on Chord# (Schütt et al., 2007) using a transactional layer implementing Paxos consensus algorithm (Moser & Haridi, 2007). The real Wikipedia runs on a server farm with a fix amount of nodes, with a centrally-managed database. The decentralized version allows the network to add more nodes to the system when more storage capacity is needed. The stored items are symmetrically replicated, and each transaction runs its own instance of a transaction manager, preventing the system from having a single point of congestion.

To validate our implementation of the atomic transactional DHT using Paxos consensus algorithm, which is part of Trappist, running on top of the Relaxed-Ring, we decided to give the task of implementing a decentralized Wikipedia to the students of the course “Languages and Algorithms for Distributed Applications”, given at the Université catholique de Louvain, as a course for engineering and master students. The students had two weeks to develop their program having access to Beernet’s API for building their peer-to-peer network, and for using the transactional layer to store and retrieve data from the network.

To store data in a DHT, the information has to be stored as items with a key-value pair. A paragraph in an article was the granularity used to organize the information of the wiki. Articles were stored as a list of paragraphs. Using articles as the minimal granularity would have not been convenient because users never update more than one article at the time. Therefore, the transactional layer would have been used to update only an item at the time, being useful only for managing replica consistency. Furthermore, such granularity would not allow concurrent user to work on the same article. Figure 11 depicts how using paragraphs as the minimal granularity can be useful to allow concurrent users updating the same article. On the figure, both users get a copy of an article composed by three paragraphs. Each paragraph has its own version, marked as timestamps (ts). User *A* modifies paragraph 1 and 3, while user *B* modifies paragraph 2. When user *A* commits her changes, the transactional layer guarantees that both paragraph will be updated, or none of them will. This property is particularly interesting if we consider that the article could be source code of a program instead. Allowing only one change could introduce an error in the program. Continuing with the example, since modifications of users *A* and *B* do not conflict, both transactions commit successfully. Consequently, if user *B* would have also modified either paragraph 1 or 3, only one of the commits would have succeeded. It is up to the application to decide how to resolve the conflict.

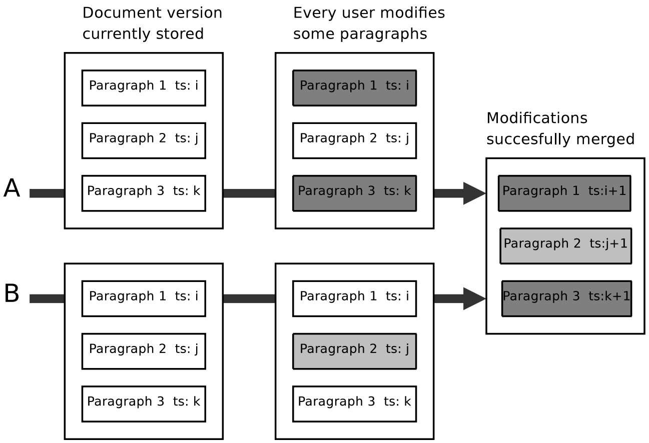

Figure 11. Users *A* and *B* modify different paragraphs of the same document. Both can successfully commit their changes because there are no conflicts.

Code 6 has been taken and modified from one of the student projects, with permission of the authors Alexandre Bultot and Laurent Herbin. The code performs several transactions so as to update the article. The modifications are divided into two lists of paragraphs, which are determined by the application: `ToCommit`, containing all paragraphs with modifications, and newly added paragraphs too; `ToDelete` are obviously the paragraphs that will be deleted. These procedures imply several calls to `write` and `remove` on the transactional object. Calling `executeTransaction` on the `Node` guarantees that all of them will be committed, or the whole update fails. This version is slightly simplified, because adding and removing items has also implications on the list of paragraphs of the article. The representation of such list is application dependent, so we will not include it on these code samples.

As we can see, reading an article and committing the correspondent updates is fairly simple using the transactional DHT API. As an average, the student projects were about 600 lines of code, including the graphical interface, and the code for bootstrapping the peer-to-peer network. The students were not asked to implement an HTML interface. Instead, they could implement a simple GUI using the Mozart programming system, to make it simpler to interact with Beernet.

The feedback from the students helped us to improve our system, and it confirmed us that the provided API is suitable for other programmers to develop applications on top of our system. The students agreed that all the complexity of building the network, routing messages, storing and retrieving data from the replicas, was well hidden behind the API. Unfortunately, they got the feeling that their student project did not let them test their skills on distributed programming for decentralized systems, because they were working on a higher level. This is of course positive for Beernet as programming framework, but we need to reconsider the project

Algorithm 6 Committing updates and removing paragraphs

```
proc {RobustCommit ToCommit ToDelete}
   Trans = proc {$ Obj}
               for UpdPar in ToCommit do
                  {Obj write(UpdPar.id UpdPar.text)}
               end
               for DelPar in @ToDelete do
                  {Obj remove(DelPar.id DelPar.text)}
               end
               {Obj commit}
            end
in
   {Node executeTransaction(Trans Client paxos)}
end
```

as an academic activity.

Future Development

We are planning to develop a system to help Linux users to efficiently choose and install packages by sharing knowledge among them. One of the targets of the application is to create an inference recommendation system that can suggest packages according to what the other users in the community are using. For instance, the majority of Java developers use Eclipse as their IDE. Therefore, if a user has installed the package for Java development, the system could suggest the installation of Eclipse. Similarly for Ruby developers, the system could suggest Ruby-on-Rails. The second target of the application is to share the knowledge acquired with respect to installability. In Linux, packages depend and have conflict with other packages. Therefore, installing one may imply to uninstall another one needed by a third one. But dependencies have also alternatives, so it is possible to find work arounds that satisfy the installation of any set of packages. Finding a solution can be highly time-consuming, but, if another user has already found the work around, the information could be shared with the rest of the community. These two goals imply a decentralized storage where every peer contributes with information. We need to further investigate about which transactional protocols are the best suitable for its design and implementation.

Conclusion

We have started this work discussing about the complexity of building dynamic distributed systems. We explained why decentralized systems overcome the disadvantages of classical centralized architectures, and we also explained that self-management is the key to deal with the higher complexity of decentralized systems. We also identified the need for maintaining the state of the system replicated across the network.

In Section , we reviewed existing solutions to build self-organized and self-healing structured overlay networks as the base for decentralized systems. We decided to use the relaxed-ring as our network topology for Beernet, because of its cost-efficient ring maintenance which does not relies on transitive connectivity. The basic DHT provided by the relaxed-ring has been improved with a replication layer built op top of it. The layer is built using symmetric replication as the replication strategy. To guarantee the consistency and coherence of the replicas, a transactional layer called Trappist is in charge of providing atomic updates of the items, with the guarantee that the majority of the replicas store the latest value. Trappist implements three different transactional protocols, which are described in Section . This layer is part of the whole implementation of Beernet, which as a whole provides a self-managing peer-to-peer network with transactional replicated DHT.

To validate the ideas presented in this paper, we showed in Section a set of applications built on top of Beernet. They take advantages of the different transactional protocols to provide synchronous and asynchronous collaborative tools. One of the applications we presented is design and implemented by a third party, validating Beernet as programming framework.

References

Collet, R., & Van Roy, P. (2006). Failure handling in a network-transparent distributed programming language. In *Advanced topics in exception handling techniques* (p. 121-140).

Dabek, F., Brunskill, E., Kaashoek, M. F., Karger, D. R., Morris, R., Stoica, I., et al. (2001). Building peer-to-peer systems with Chord, a distributed lookup service. In *Hotos* (p. 81-86). IEEE Computer Society.

Daswani, N., & Garcia-Molina, H. (2002). Query-flood dos attacks in gnutella. In *Ccs '02: Proceedings of the 9th acm conference on computer and communications security* (p. 181-192). New York, NY, USA: ACM.

FreeNet Community. (2003). *The freenet project.* `http://freenetproject.org`.

Ghodsi, A. (2006). *Distributed k-ary system: Algorithms for distributed hash tables.* Unpublished doctoral dissertation, KTH — Royal Institute of Technology, Stockholm, Sweden.

Gnutella. (2003). *Gnutella.* `http://www.gnutella.com`.

Gray, J., & Lamport, L. (2006). Consensus on transaction commit. *ACM Trans. Database Syst.*, *31*(1), 133-160.

Grolaux, D., Mejías, B., & Van Roy, P. (2007, September). PEPINO: PEer-to-Peer network INspectOr. In M. Hauswirth, A. Wierzbicki, K. Wehrle, A. Montresor, & N. Shahmehri (Eds.), *The seventh ieee international conference on peer-to-peer computing* (p. 247-248). IEEE Computer Society.

Hewitt, C., Bishop, P., & Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *Proc. of the 3rd ijcai* (p. 235-245). Stanford, MA.

Krishnamurthy, S., & Ardelius, J. (2008). *An analytical framework for the performance evaluation of proximity-aware structured overlays* (Tech. Rep.). Swedish Institute of Computer Science (SICS), Sweden.

Markatos, E. P. (2002). Tracing a large-scale peer to peer system: an hour in the life of gnutella. In *2nd ieee/acm international symposium on cluster computing and the grid.*

Mejías, B., & Van Roy, P. (2008). The relaxed-ring: a fault-tolerant topology for structured overlay networks. *Parallel Processing Letters*, *18*(3), 411-432.

Mesaros, V., Carton, B., & Van Roy, P. (2005). P2PS: Peer-to-peer development platform for mozart. In P. V. Roy (Ed.), *Moz* (Vol. 3389, p. 125-136). Springer.

Miller, M. S., & Shapiro, J. S. (2003, December). Paradigm regained: Abstraction mechanisms for access control. In V. Saraswat (Ed.), *Asian'03.* Springer Verlag.

Moser, M., & Haridi, S. (2007). Atomic Commitment in Transactional DHTs. In *Proceedings of the coregrid symposium. coregrid series.* Springer.

Mozart Consortium. (2008). *The mozart-oz programming system.* `http://www.mozart-oz.org`.

Plantikow, S., Reinefeld, A., & Schintke, F. (2007). Transactions for distributed wikis on structured overlays. In *Managing virtualization of networks and services* (p. 256-267). Available from `http://dx.doi.org/10.1007/978-3-540-75694-1\_25`

PostgreSQL Global Development Group. (2009). *PostgreSQL: The world's most advanced open source database.* `http://www.postgresql.org/`.

Ratnasamy, S., Francis, P., Handley, M., Karp, R., & Schenker, S. (2001). A scalable content-addressable network. In *Sigcomm '01: Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications* (p. 161-172). New York, NY, USA: ACM.

Rhea, S., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., et al. (2005). *Opendht: A public dht service and its uses.* Available from `citeseer.ist.psu.edu/rhea05opendht.html`

Ripeanu, M., Foster, I., & Iamnitchi, A. (2002). Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system. *IEEE Internet Computing Journal*, *6*, 2002.

Rowstron, A., & Druschel, P. (2001a). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, *2218*, 329.

Rowstron, A., & Druschel, P. (2001b). *Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility.* Available from `citeseer.ist.psu.edu/rowstron04storage.html`

Schütt, T., Schintke, F., & Reinefeld, A. (2007). A structured overlay for multi-dimensional range queries. In *Euro-par 2007.*

Schütt, T., Schintke, F., & Reinefeld, A. (2008). Scalaris: reliable transactional p2p key/value store. In *Erlang '08: Proceedings of the 7th acm sigplan workshop on erlang* (p. 41-48). New York, NY, USA: ACM.

Stoica, I., Morris, R., Karger, D., Kaashoek, F., & Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 acm sigcomm conference* (p. 149-160).

Wikimedia Foundation. (2009). *Wikipedia, the free encyclopedia.* `http://en.wikipedia.org/wiki/Wikipedia`.

Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D., & Kubiatowicz, J. D. (2003). Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*.