A Transactional System for Structured Overlay Networks

Valentin Mesaros Raphaël Collet Kevin Glynn
Peter Van Roy
Université catholique de Louvain
2 place Sainte Barbe, B-1348 Louvain-la-Neuve, Belgium
{*valentin,raph,glynn,pvr*}@*info.ucl.ac.be*

March 14, 2005

Abstract

We propose a system for executing transactions on top of structured peer-to-peer networks. Our system guarantees most properties usually expected from transactions, namely atomicity, independance, and consistency. The system is based on distributed locking, with a fully decentralized management. It avoids deadlocks and starvation by assigning priorities to transactions.

Contents

1 Introduction

In this document we propose a transactional system for fully distributed, structured peer-to-peer (P2P) networks. We are interested in running transactions in a dynamic decentralized system, where nodes could fail with a relative high rate, and where data items could be temporary inaccessible. To our knowledge, currently there exists no system working in such a dynamic environment that provides all the ACID properties of transactions. The only systems that provide ACID properties are more or less centralized, and limited in the number of participating sites.

In this paper we extend a lightweight transaction protocol and provide an architecture implementing a robust, full transactional system on top of structured overlay networks. The system can be implemented as a service that runs on top of P2PKit [1, 2]. The latter provides a platform for services, and uses our P2PS [3, 4] library for its underlying P2P network.

Our transactional system can be seen as a lightweight group management protocol. Given a large set of objects distributed over an overlay network, a transaction consists of a (usually small) dynamic number of objects that are temporarily collaborating to perform the transaction. The transactional protocol provides an atomic multicast with rollback capability for any number of such temporary groups concurrently. We therefore see our transactional system as a natural service for applications written on top of decentralized overlay networks.

The document is organized as follows. We present the algorithms we use and some principles behind our transactional model in Section 2. In Section 3 we describe the system architecture, identifying the main components of our system. In Section 4 we specify how a transaction is executed in our system. Failures are analyzed in Section 6. Section 7 compares our approach with related works.

2 Algorithms

This section presents a simple yet practical transaction system, based on two-phase locking [5]. We discuss the issues to deal with when distributing that system. From the discussion we state a few principles to follow.

2.1 A Centralized Transaction System

In our system we consider transactional agents that invoke objects. The agents use the two-phase locking technique for locking objects. A transaction has two phases: the "growing" phase, where the agent acquires locks and does not release them until the "shrinking" phase, where it releases locks without acquiring new ones. This guarantees that all transactions are serializable. In order to avoid so-called cascading aborts, we use the variant called strict two-phase locking. In this refinement, all locks are released simultaneously at the end of the transaction. However, such a simple scheme does not prevent deadlocks. We avoid them by introducing transaction priorities.

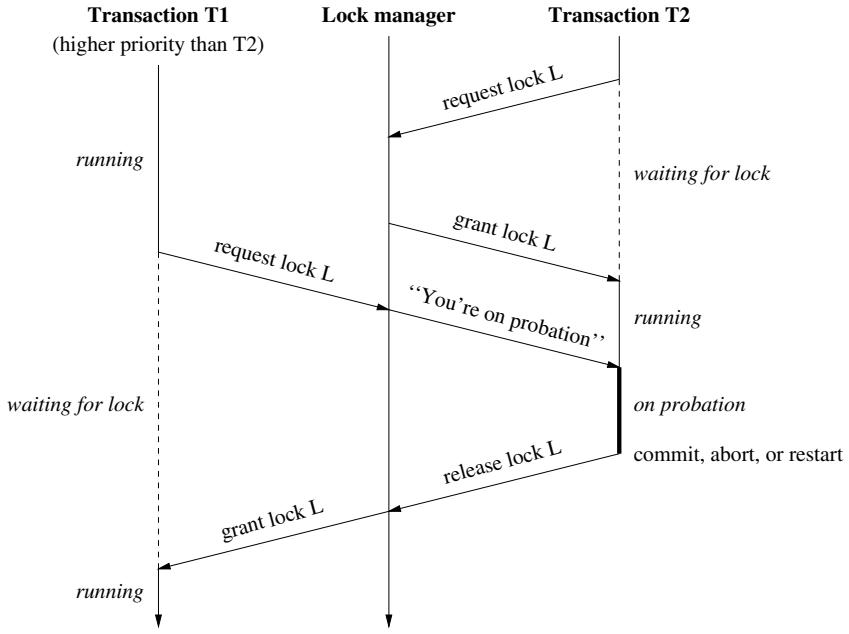

Figure 1: Example of two transactions competing for a lock. Transaction T2 is marked as being on probation.

Priorities and Probation. A deadlock happens when we have a cycle of transactions where each transaction waits for a lock that is held by the next one in the cycle. In order to break the cycle, each transaction is assigned a *priority*. Newer transactions are given lower priority than older ones. If two or more transactions wait for a lock, the lock will be granted to the highest priority transaction first. Older transactions are therefore guaranteed to eventually get the lock. A simple implementation of priorities uses timestamps.

When a transaction T_1 waits for a lock that is held by a transaction T_2 with lower priority, T_2 is marked as being *on probation*. This means that transaction T_2 must complete without acquiring new locks. In other words, it forces T_2 in the shrinking phase of two-phase locking. The transaction can still continue and eventually commit, provided no more locks are required. This is an optimistic strategy for concurrency control. If T_2 waits for a lock or asks for a new lock, then it is aborted, which causes all its locks to be released. This breaks the deadlock cycle, if there was one. Possibly, transaction T_2 can be restarted with the same priority. Figure 1 shows a scenario with message exchanges for the example. Every object in the system has a lock manager that is responsible for controlling its lock. This entity can also be responsible for cancelling the changes made in the object by an aborted transaction. The probation phase of transaction T_2 is shown as a bold line.

Note that this scheme also prevents livelocks and starvation. At one point, if the

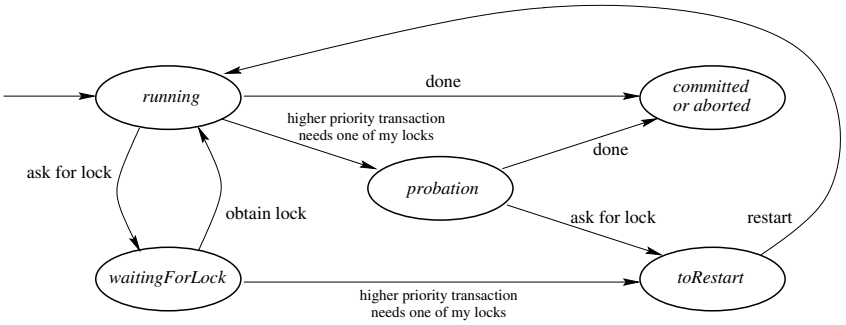

Figure 2: Finite state automaton for a transaction

system has n transactions that have higher priority than T_2, then T_2 may be restarted at most n times. Therefore, T_2 will eventually complete, provided all transactions terminate.

States of a Transaction. Figure 2 depicts a finite state automaton for a transaction. When started, a transaction enters state *running*. In this state, objects are invoked and computations are performed. When the transaction terminates, it goes to state *committed or aborted*. Abortion indicates that the transaction cannot succeed, and is independent from lock management. When a lock is needed, the transaction sends a request and enters state *waitingForLock*, going back to *running* whenever the lock is granted. When a higher priority transaction asks for a lock, the transaction either goes on *probation* if it was running, or *toRestart* if it was waiting for a lock. The state *toRestart* indicates that the transaction has been stopped to avoid a deadlock, and should be restarted, going back to *running*. Execution in state *probation* is similar to *running*, except that asking for an extra lock immediately leads to state *toRestart*.

2.2 Making the Algorithm Distributed

The transactional system we described can easily be adapted to a distributed setting, provided a few issues are addressed. We have chosen to address distributed settings where the objects are not mobile, and each lock is managed where its corresponding object is located. This is a very common setting today, and it allows simple reasoning about failures and network behavior.

In such a system, a transaction may acquire locks and invoke objects at different sites. Three major issues may arise here. First, we have to deal with a fully decentralized lock management. The presence of a deadlock is a global condition on the system. Second, several sites can be involved in a transaction. In order for the transaction to be atomic and consistent, all those sites must reach the same decision (commit, abort, or restart). Third, sites may fail.

The first and third issues are discussed below. We address the second issue by making one agent in the system responsible for taking the decision. The other sites

should conform to its decision. This implies, for instance, that before committing, the agent taking the decision must ensure that all sites involved in the transaction have completed and are ready to commit.

Probation. Putting a transaction on probation must be done properly, otherwise deadlocks may occur. Let us illustrate the problem with an example. Two transactions T_1 and T_2 are concurrently locking objects O_1 and O_2, which are located at different sites. We assume that T_1 has higher priority than T_2. The situation is the following. Transaction T_1 has locked O_1, and asks for locking O_2. The object O_2 is locked by T_2, which asks for locking O_1. At the site of O_2 it is known that transaction T_2 goes on probation. At the site of O_1 transaction T_2 is waiting for a lock. One must put both informations together in order to decide to restart the transaction.

The natural extension of the optimistic strategy in the distributed setting is to inform all sites involved in the transaction that the transaction is on probation. If the transaction was waiting for a lock at a site, it releases all locks, and restarts. Otherwise the transaction continues. Likewise, if a lock is required when the transaction is on probation, the transaction is restarted.

Note that pessimistic strategies are possible, too. One might consider highly probable that a site involved in the transaction is waiting for a lock, or will ask for a lock soon. The site that was first put on probation then immediately makes the transaction restart. This strategy might be efficient for transactions involving many objects. Both kinds of transactions may even coexist in the system.

Priorities. The priorities assigned to transactions at the various sites must be somewhat coordinated. A new transaction must have lower priority than older ones. So the site that initiates this transaction should not give it a priority that is too high with respect to existing transactions in the system.

If priorities are implemented by timestamps, each site may adapt its clock to keep it close to other sites' clocks. The generation of timestamps must be increasing, therefore it is easier to adapt the clock by putting it forward. Here is a simple algorithm. When a site gets to know a transaction, it compares the timestamp of that transaction with its own clock time. If the timestamp is greater, it puts its clock forward such that its own future timestamps will be strictly greater. One must also avoid identical timestamps. A simple way to solve this is to append a site-specific number to the timestamp.

Site failures. Suppose a site is involved in a transaction, and that it fails. We assume here that the other sites will eventually detect the failure. What should they do then? It depends on which site has failed, and in which state the transaction was. But the important thing is that they must act consistently.

If the site held the agent responsible for deciding the transaction outcome, the other sites should find out whether a decision was taken before the failure. If so, they must all be informed of the decision and conform to it. Otherwise, they're on their own. Whatever they do, they must do the same. They can safely abort, for instance. Or they can elect a new agent for deciding the transaction's outcome.

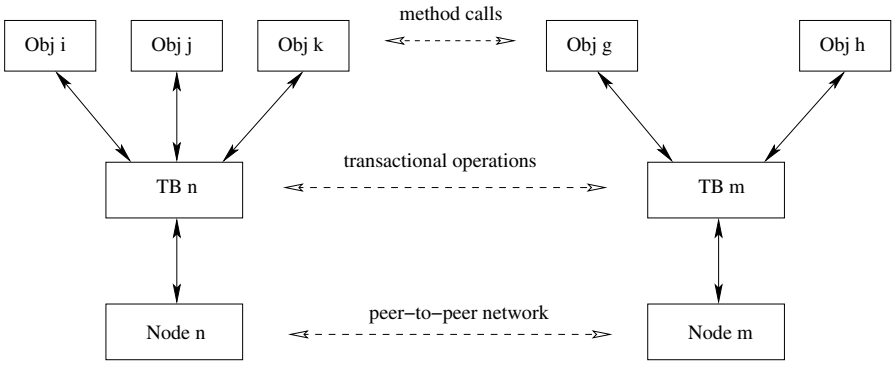

Figure 3: Architecture of the transactional system.

If the other cases, the decision depends on the transaction's responsible agent. If the transaction had still to execute on this site, then the transaction should abort or restart. Restarting the transaction is useful if the objects on the failed site were replicated. Note that the replication mechanism must restore the object in the state it was before the transaction. If the failure detection happens after the agent took a decision, then the other sites must terminate normally. Whatever the decision, the system's state should be as if the failure happened right after the transaction's termination.

3 System Architecture

Our system is composed of three kinds of components: objects, transaction blocks, and overlay nodes. They correspond to three working levels: data (or object) level, transaction level, and overlay network level, respectively. Figure 3 shows a simplified snapshot of the architecture of our system, emphasizing the relation between the components locally as well as remotely. In our example we have two overlay nodes, Node n and Node m, each being associated its own transaction block TB n and TM m, respectively. We have three objects at node n, while node m contains two objects.

3.1 The Object Level

At the data level one would read and modify data items. In our case, this will be done by calling object methods, regardless whether the objects are local or located remotely. These calls will be executed within the boundaries of well-defined transactions. For executing a transaction, the transaction level is invoked.

Each object is associated a unique identifier *Oid*. Given its *Oid*, the object is stored within the system at an overlay node associated to that identifier. This association is explained below. The object identifier allows to refer an object independently from its location in the overlay network.

3.2 The Transaction Level

In our model, there exists one transaction "block" per node. The transaction block is composed of one transaction service (TS), transaction managers (TM), and transaction participants (TP). A transaction block collaborates with other transaction blocks to execute transactions.

Each transaction is executed by one transaction manager TM and several transaction participants TP's. Transaction managers and participants are created by the transaction services at each site involved in the transaction. The TM is located at the overlay node where the transaction was initiated. The main role of a TM is to coordinate the execution of its transaction. The transaction has one transaction participant TP at each overlay node involved in the transaction. The TP is the local representative for the transaction on its overlay node. It collaborates with its TM, and is responsible for the interaction with objects involved in the transaction. A more detailed description is given in Section 4.

3.3 The Overlay Network Level

The underlying overlay network is responsible for two things. First, it provides access to items within the system. The overlay nodes should organize the network to make this access scalable and reliable, even when nodes join or leave the network. Those P2P networks behave like distributed hash tables (DHT). Second, it provides efficient communication primitives.

Data replication is mostly orthogonal to transactions. The connection between the two will be discussed later. For the sake of simplicity, an object exists only at one overlay node, namely the node associated to its object identifier *Oid*. Therefore, when the site of an overlay node fails, all the objects stored on this site are lost. Accessing those objects will return an error.

Assumption 1 *There is no object replication mechanism in the system.*

Nevertheless, we use a very simple replication scheme for the state of transaction managers. This redundancy will pay back when the TM of a transaction is lost in the failure of its site. As we will show in Section 6, another node can carry on the interrupted transaction.

The overlay network provides message-passing communication primitive.

Assumption 2 *The communication provided by the overlay network is reliable and ordered.*

That is, while the nodes exchanging messages are up, all messages from one node to another eventually reach their destination in order. If a communication end node fails, the failure is eventually notified to the other end of the communication link, possibly after a timeout.

An example of a DHT is Tango [6]. It is implemented by the library P2PS [4, 3], which has been developed at UCL. We intend to use it for implementing our transaction system.

4 Transaction Protocol

In Section 2 we described the main algorithm and principles of our transactional system. This section gives in a detailed manner how we actually implement this algorithm as a protocol.

Recall that our transactional system uses the strict two-phase locking (2PL) scheme. The two phases correspond to, first, acquiring the lock for a data, and second, after operating on the data, releasing all the acquired locks at once. For the sake of simplicity, we consider that the locks in our system are exclusive. That is, we apply the same policy to all the locks within a transaction. The system, however, can be extended to work with other types of locks (read-only, etc).

4.1 Actors of a Transaction

In our implementation we make a separation of concerns, and implement transactions with transaction service nodes, transaction managers, and transaction participants, as sketched in the previous section. Below we describe the role of each kind of entity, then we describe how they interact with each other in order to execute the transaction.

Transaction Service. Each overlay node in our system is associated a transaction service TS. The transaction service (a) creates the transaction managers and participants for transactions involving objects on this site, and (b) manages the access to objects.

Transaction Manager. The manager of a transaction is taking care of the whole execution of the transaction. In our model there is one TM per transaction, which located at the site where the transaction started. The transaction manager (a) initiates the actual transaction procedure, and (b) manages the transaction, taking the decision to eventually commit, abort, or restart the transaction.

Note that the TM does not invoke objects itself, nor does it manages the transaction locks. Instead it interacts with all the transaction participants on all sites involved in the transaction.

Transaction Participant. A transaction participant is a process related with one or more objects involved in its transaction. For a given transaction, there should be one TP located at each overlay node that holds an object involved in that transaction. The transaction participant (a) acquires and releases locks to access the objects involved in its transaction, and (b) invokes those objects.

4.2 Transaction Execution

At the object level, a transaction is simply a sequential algorithm involving method calls on objects, let us call it a *transaction procedure*. The actual transaction takes place at the transaction level, where the transactional protocol runs. Upon completion, a transaction returns either *committed*, indicating its success, or *abort*, indicating its failure.

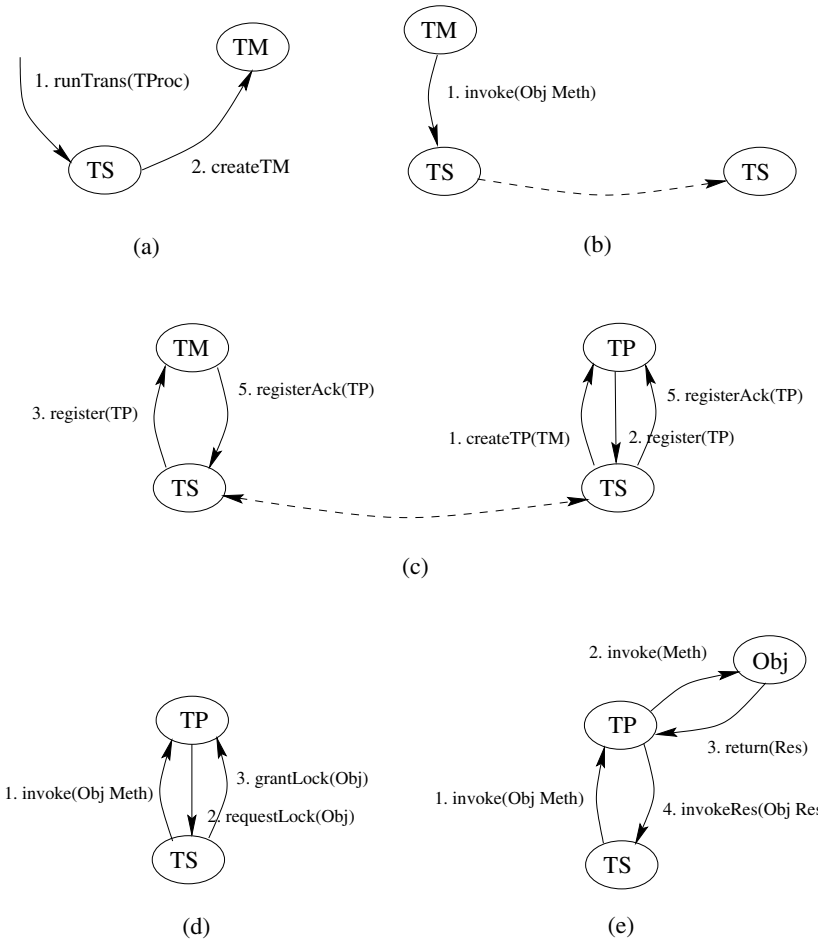

Figure 4: Different execution phases of the transaction, and the components involved.

Figure 4 depicts the main phases of the transaction execution, and the interaction between components.

(a) A transaction begins when the TS is asked to run a transaction procedure. In response to this, it creates a TM instance.

(b) The TM then runs the transaction procedure, and emit method calls on objects. A method invocation request is transmitted to the TS associated with the overlay node where the object is.

(c) If no TP exists on that overlay node for the transaction, a TP instance if created by the TS on that node. Once created, the TP registers to the transaction's TM.

(d) The TP is then asked to invoke the object. It first requests the lock of the object to the local TS.

(e) Once the lock is granted, the TP invokes the method of the object, and sends the result back to the transaction's TM, via the local TS.

Object invocations might emit other method calls, possibly on remote objects. Each invocation to a remote object is passed to the local TS, which contacts the TS on the object's overlay node. The TS will treat this invocation as if it came from the TM of the transaction. It will therefore create a TP instance if necessary. The new TP must register to the transaction's manager. The invocation result is then sent back to the object that issued the method invocation.

We continue our presentation by giving more details on the transaction execution. We split the presentation into two parts, describing the transaction execution at the TM and TP sides.

Transaction Execution at the Transaction Manager. Figure 5 presents the finite state automaton of a transaction manager. A TM can be in four possible states: *running*, *probation*, *committed or aborted*, and *toRestart*. These states correspond to the transaction states described in Section 2. Note that a TM does not have the state *waitingForLock*, since it does not directly access objects.

When a TM is created by the TS, it directly goes in state *running*. This is the state where the main transaction activity takes place. In this state, the TM requests the invocation of methods, and accepts TP registering. The TM leaves state *running* in three different conditions.

- When it receives the message onProbation from one of its TP's, it propagates the message to all the other TP's, and then goes to state probation.
- When the whole transaction procedure has been executed, and corresponding results gathered, it sends the message commit to all TP's, and goes to state *committed or aborted*.
- When it receives abort from a TP, it propagates the message to all the other TP's, and goes to state *committed or aborted*.

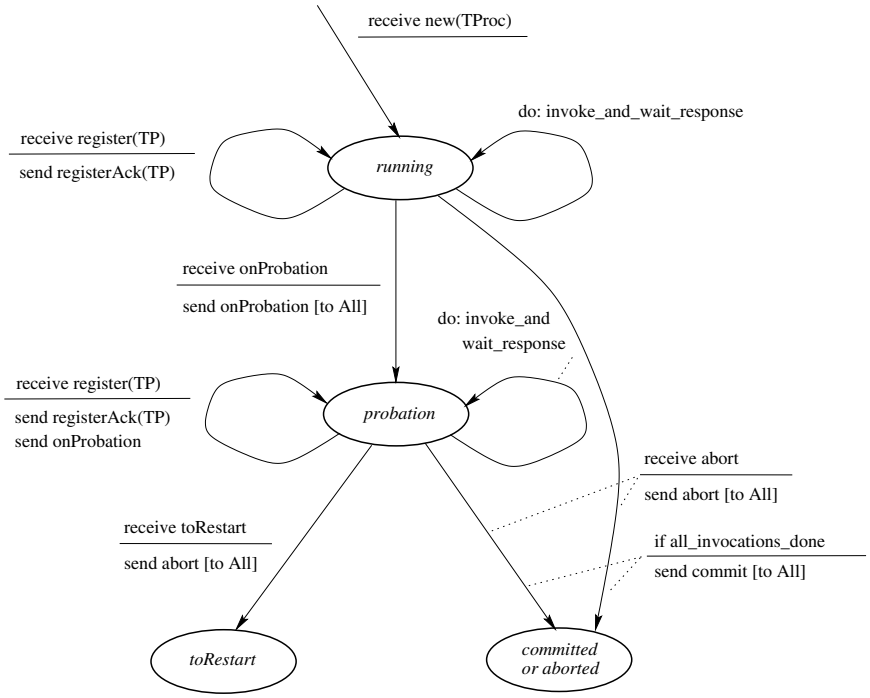

Figure 5: Finite state automaton of a transaction manager

As in state *running*, in state *probation* the TM can request for method invocation, and accept the registration of new TP's. Note that, in order to be consistent, the new TP's will be sent the message onProbation. Here again, receiving the message abort, as well as the termination of the transaction procedure are dealt with as in state *running*. A new condition may, however, appear here. Receiving the message toRestart makes the TP's abort, such that the transaction can be restarted.

Transaction Execution at a Transaction Participant. Figure 6 shows the finite state automaton of a transaction participant. A TP can be in six possible states: *registering*, *running*, *waitingForLock*, *probation*, *committed or aborted*, and *toRestart*. These states mainly correspond to the transaction states described in Section 2. Note the existence of the state *registering*, which reflects the TP registration to its manager.

Once a TP is created by the local transaction service, it immediately sends a registering message, and enters state *registering*. In this state the TP must make sure that it is known by its manager, before attempting any object invocation. After being registered, the TP goes in state *running*, where it is asked for invoking methods on local objects. To this end, it must lock each invoked object via its TS, temporary going to state *waitingForLock*.

From state *running*, the TP can carry on either with going to state *probation*, or with passing in state *committed or aborted*. The condition to pass in state *probation* is fulfilled when the TP receives the message onProbation from the local TS, indicating that there is another transaction with a higher priority that wants to access one object locked locally by this TP. If the TP receives the message commit from its TM, indicating that the transaction has succeeded, it goes to state *committed or aborted*. If a method invocation returns an error, or the TP receives the message abort from its manager, indicating the failure of the transaction, it also goes to state *committed or aborted*.

As in state *running*, in state *probation* the TP can invoke methods on local objects. The difference, however, lays in the fact that in state *probation* the TP is not allowed to acquire new locks. As explained in Section 2, this avoids deadlocks. Asking for an extra lock immediately leads to state *toRestart*. In this case, the TP sends the message toRestart to its TM. Otherwise, the transaction carries on, and the TP eventually ends up in state *committed or aborted*. The transaction is committed or aborted depending on whether the TP receives the message commit, abort, or a method invocation returns an error.

5 Working on an Overlay Network

Given its highly dynamic topology, a P2P system is known to be difficult to work with. However, such a system provides nice properties such as decentralization, scalability, and availability. In this section we describe the particularities of working on top of structured overlay networks, such as Tango [6], Chord [7], and Pastry [8]. We enumerate the main advantages of such systems and explain how one can exploit them, as well as some challenges of working with such systems.

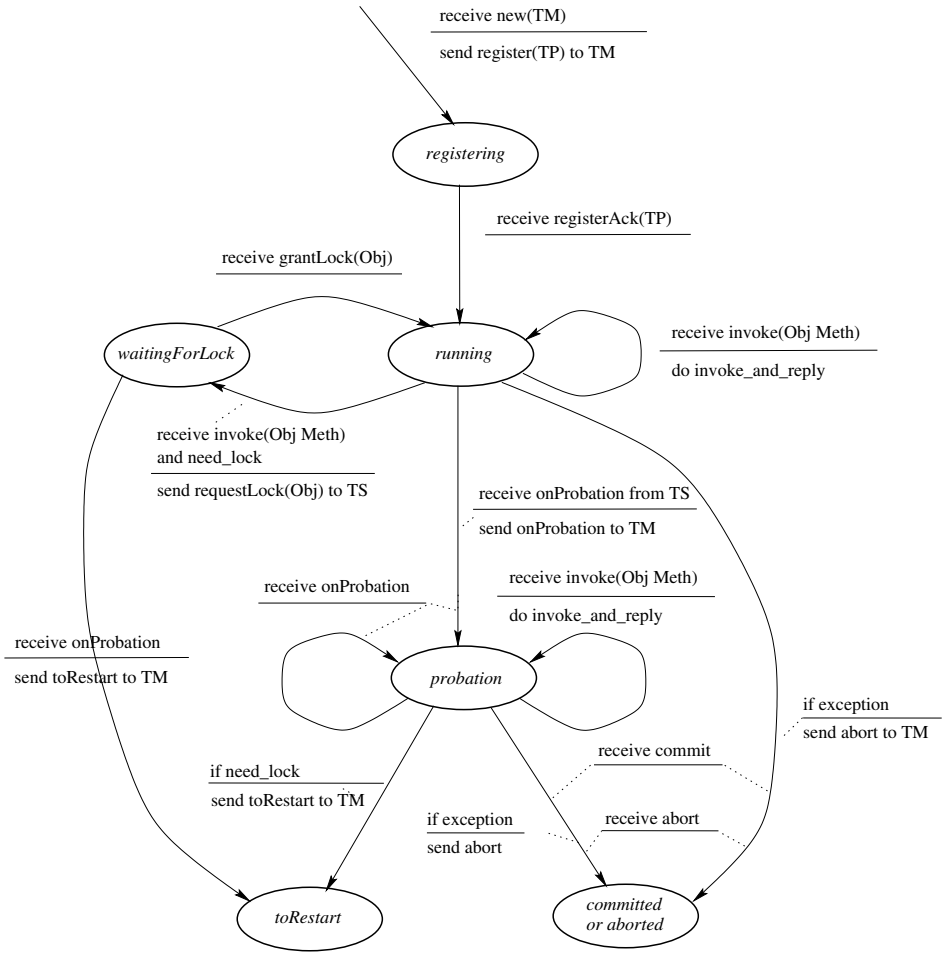

Figure 6: Finite state automaton of a transaction participant

5.1 Advantages

Location Independent Communication. One process on a site can communicate with a process on another site without knowing the latter's network address. One use of this property is that nodes can physically move from one physical network domain to another while keeping the same identity within the P2P system. A practical example is the fact that in our transactional system, the fault tolerance can be made transparent. This is possible since the overlay nodes hosting the transaction participants of a transaction are not known by their IP addresses, but merely as responsibles of the objects within that transaction. When one of the nodes fails, another one can take over as the new responsible of its objects.

Network Scalability and Efficient Communication. Generally speaking, a structured overlay network offers a way to organize a very large number of nodes in a network where the communication between nodes is efficient. For instance, in a network of N nodes, each node in the network has $O(\log N)$ neighbors, and any message sent from it to any other node should not take more than $O(\log N)$ hops. This quality of service is very useful in a transactional system involving a large number of nodes, contributing to the scalability of the transactional system itself. Beside the one-to-one communication primitives, a P2P system can also offer efficient group communication primitives. These primitives are very useful in a transactional system where group communication is the main primitive used.

Availability. Since within a P2P system there is no one site controlling the well-functioning of the system, the problem of single node failures is inherently solved, thus naturally increasing the availability of the system. Moreover, any node within a P2P system can serve as an entry point to the system. That is, in order to enter the system, it is enough to know one node within the system. Speaking about service and data availability, in a distributed hash table, each node has a well-defined responsibility. When a node fails or leaves, its responsibility is passed to another node. With little effort a fault tolerant schema can be applied, thus increasing the service/data availability of the system.

5.2 Challenges

Dynamic Topology. In a P2P network nodes join and fail with a relatively high rate. Working in such an environment is tricky. For instance, it is almost impossible, or at least impractical, to obtain an overall snapshot of the large and dynamic network. Applications working with such networks should consider node failure to be a highly probable event and deal with it accordingly.

Multihop Communication. While in a classical client/server model, the communication is done point-to-point, in a P2P system the communication between two nodes can pass through $O(\log N)$ other nodes within the system, where N is the maximum number of nodes in the system. This involves network delays and might decrease the

communication reliability. The reason for this is the fact that when a node fails along the path between two collaborating nodes, some messages may be lost. Remember, since a structured P2P system is self-organized, the problem will be temporary. But a little overhead must be added in order to guarantee reliable communication between nodes.

6 Dealing with Failures

After seeing how the transactional system behaves in an environment without failures, we are now interested in the behavior of transactions face to failures. For our system we are interested in two types of failures: node failures at the overlay network level, and what we call "object access failures". The former type of failure can occur relatively often in a dynamic system as an overlay network, and it imminently implies process failures. The latter type of failure occurs when the access to an object within a system fails. We continue this section with describing the two types of failures.

6.1 Node Failures

Usually, information redundancy is used during the execution of a transaction, allowing its recovery if failures occur. In order to be able to recover, a transaction manager can flush information about the transaction status to a replica storage. That is, the information is replicated to other nodes to be taking over if the primary node fails.

Given that in our system, for simplicity, we only replicate the transaction's state, for a transaction we will possibly need to run a recovery protocol only for the TM. The transaction execution at a TP site that fails is not going to be recovered. It is interesting, however, to see how a TM should react face to the failure of its TP's.

Transaction Manager Failure. We assume that the failure of a transaction manager is caused by the failure of the overlay node it is associated with. In this situation the overlay network eventually notices the node failure, and reorganizes itself accordingly. This results in automatically assigning a new responsible of the transactions initiated at that site. The new responsible will be among the nodes we have replicated information about the status of the running transactions. The recovery procedure at the new TM site should read the log file, and decide whether to carry on with the current transaction, or to simply abort it. For our system we propose the following strategy.

- If the transaction is in state *running* or *probation*, and we have gathered the results for all the invocations, we continue with sending the message commit to all the TP's we know of. At this stage we know that all TP's (except the failed ones) are ready to commit.
- In all the other cases, the transaction is going to be aborted. The TM sends the message abort to all the TP's it knows about. The reason for this is to avoid any possible inconsistent state. Receiving abort, a TP releases all locks it detains, and destroy all the temporary workspaces.

Besides ensuring the ACID properties for transactions, we are also concerned with the system's liveness. The recovery procedure for the transaction manager site allows us to release the locks acquired beforehand. Nevertheless, since we are considering a large, dynamic, and long-lived system, we introduce an additional precaution for not causing starvation. Thus, each lock provided by an object is associated a temporal lease (a relative long period). The lease is renewed at each contact between the TM and the corresponding TP. If during a transaction, the TM site does not contact a peering TP, the corresponding lease will eventually expire, and so will the associated locks. This preserves the liveness property of the system; for instance, in the case where the replication degree of the system would be overtaken by the failure grandeur.

Transaction Participant Failure. During a transaction it could happen that one or more TP sites fail. That is, they are not accessible anymore, either temporary or permanently. For our system, we make the following assumptions.

Assumption 3 *After a certain period of not being accessible, an overlay node is considered to be failed, and removed from the overlay network.*

Assumption 4 *Two overlay nodes always eventually agree on the failure status of a third node in the network.*

Here we are interested in how a transaction manager should react face to TP failures. Depending on the transaction status, the recovery procedure at the TM site when one or more TP sites fail differ. We assume that

Assumption 5 *The objects stored at a failing node eventually disappear, together with any temporary working spaces.*

For our model we adopt the following strategy. Note that this is very close to the procedure of dealing with TM failure.

- If the transaction is in state *running* or *probation*, and we have completed the transaction procedure, we continue with sending commit to all the TP's we know about. At this stage we know that all TP's (except the failed ones) were ready to commit. Given Assumption 5, we don't need to care here about the TP's that might have just failed. No inconsistent state is observable from the failed TP's.
- In all the other cases, the transaction is going to be aborted, or restarted. The TM sends a message abort to all the TP's it knows about. All the locks acquired for this transaction will be released.

6.2 Object Access Failure

Another type of failure we might see in our system is object access failure. This failure occurs when the object to be accessed is not present in the system.

In our system, the overlay network is in charge of routing all messages to their destination. A message addressed to an object with identifier *Oid* will be delivered to the overlay node responsible of *Oid*. If the corresponding object is not stored by that node, an error message will be issued. If this happens to a transaction invoking an object, the transaction simply aborts.

7 Related Work

The Global Store. There exists quite a few transactional systems. The Global Store [9] can be viewed as an early attempt of our team towards distributed transactions. It was directly implemented on the distribution layer of Mozart, and does not benefit from some services offered by overlay networks. In the Global Store, each site has a copy of the shared store, and performs transactions on this store. When a transaction succeeds, the changes are replicated to the other sites. The application can be notified each time the store has changed. The Global Store uses locks, but the management of lock is centralized.

Transactions on Overlay Networks. Ivy [10] is a multi-user read/write peer-to-peer file system, with no centralized or dedicated components. Ivy is a transaction-like system built on top of the Chord [7] P2P network, and it consits of a set of logs; one log per participant. The logs are stored within the P2P network and they contain version vectors that allow Ivy to detect conflicting updates. Each participant has to maintain a private snapshot summarizing the file system state as of a recent point in time. This arrangement allows Ivy to maintain meta-data consistency without locking. However, it does not guarantee write-read consistency and it is suitable only for small groups of cooperating participants.

CFS [11] is another file system built on top of a distributed hash table-based system. CFS is simple and efficient but it is limited to read-only storage.

Memory Transactions. We would like to mention interesting ideas presented in a recent article [12]. The authors propose extensions to *software transactional memory*, that resolve someshortcomings of the latter. They first define an operation for programming blocking transactions. They then show how to compose transactions in sequence and as alternatives. This gives an excellent support for genuine *nested transactions*.

The implementation seems quite effective for memory transactions. It does not rely on locks, because lock-based programs do not compose. The execution of an atomic block of operations uses a *transaction log*, which records the memory updates that are performed. When the block completes, its log is *validated*, to check that it has seen a consistent view of memory. It then commit its changes to memory. If the validation fails, i.e., another thread has concurrently modified the memory, the block is re-executed from scratch.

We believe that such a system is not suitable for a distributed setting. The validation operation must be atomic, which implies a synchronization between the sites involved in a transaction. Moreover, this kind of synchronization is optimistic. It does not prevent starvation, for instance.

8 Conclusion

We demonstrate in this paper that implementing transactions on a structured peer-to-peer network is possible. We have designed a lightweight transaction protocol that takes full advantage of the properties of structured overlay networks. Our system uses

a fully decentralized lock-based management, and avoids deadlocks, livelocks, and starvation, at little price in performance. The system ensures the ACID properties of transactional systems. Moreover, it deals gracefully with failures.

We intend to implement this protocol as a service that runs on top of P2PKit [1, 2]. The latter provides a platform for services, and uses our P2PS [3, 4] library for its underlying overlay network.

Acknowledgements

This research was partially funded by the PEPITO project (European Union Fifth Framework Programme, Global Computing IST-2001-33234) and the EVERGROW project (European Union Sixth Framework Programme, Project number 001935).

References

[1] Kevin Glynn. Extending the oz language for peer-to-peer computing. PEPITO project deliverable D3.7, IST-2001-33234, December 2004. Available at http://www.sics.se/pepito/deliverables.html.

[2] Kevin Glynn. P2PKit Library, October 2004. Available at http://renoir.info.ucl.ac.be/twiki/bin/view/INGI/P2PKit.

[3] Valentin Mesaros, Bruno Carton, and Peter Van Roy. P2PS: Peer-to-peer development platform for Mozart. In *Proc. of the 2nd International Mozart/Oz Conference – MOZ 2004*, October 2004.

[4] Bruno Carton and Valentin Mesaros. P2PS: Peer-to-Peer System Library, October 2004. Available at http://www.mozart-oz.org/mogul/info/cetic_ucl/p2ps.html.

[5] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, March 2004.

[6] Bruno Carton and Valentin Mesaros. Improving the Scalability of Logarithmic-Degree DHT-based Peer-to-Peer Networks. In *Proc. of EUROPAR*, August–September 2004.

[7] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM*, August 2001.

[8] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. of the International Conference on Distributed Systems Platforms*, November 2001.

[9] Mostafa Al-Metwally. *Design and Implementation of a Fault-Tolerant Transactional Object Store*. PhD thesis, Al-Azhar University, Cairo, Egypt, December 2003. Available at http://www.info.ucl.ac.be/people/PVR/Metwallythesis.

[10] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-Peer File System. *SIGOPS Oper. Syst. Rev.*, 36(SI):31–44, 2002.

[11] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Wide-area cooperative storage with CFS. In *Proc. of the ACM Symposium on Operating Systems Principles*, October 2001.

[12] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. Submitted to PPoPP 2005, December 2004.