An Architecture for Dynamic Scalable Self-Managed Persistent Objects

Emmanuelle Anceaume[1], Roy Friedman[2], Maria Gradinariu[1], and Matthieu Roy[1]

[1] IRISA, Campus de Beaulieu, 35042 Rennes CEDEX, France.
{anceaume,mgradina,mroy}@irisa.fr.
[2] Computer Science Department, The Technion, Haifa 32000, Israel.
roy@cs.Technion.AC.IL

Abstract. This paper presents a middleware architecture and a generic orchestrating protocol for implementing persistent object operations for large scale dynamic systems in a self-managing manner. In particular, the proposed solution is fully distributed, allows dynamic changes in the environment, and nodes are neither assumed to be aware of the size of the system nor of its entire composition.
The architecture includes two *modules* and three *services*. The modules are expected to be instantiated and executed among relatively small sets of nodes in the context of a single multi-object operation (operation that spans multiple persistent objects) and therefore, can be implemented using known classical distributed computing approaches. On the other hand, services are long lived abstractions that may involve all nodes and should be implemented using known peer-to-peer techniques. The main contribution of the paper is to provide, for the first time, an architecture that brings together several seemingly distinct research areas, namely distributed consensus, group membership, notification services (publish/subscribe), scalable conflict detection (or locking), and scalable persistent storage. All these components are orchestrated together in order to obtain (strong) consistency on an *a priori* unsafe system.
This paper also promotes the use of *oracles* as a design principle in implementing the respective components of the architecture. Specifically, each of the modules and services are further decomposed into a "benign" part and an "oracle" part, which are specified in a functional manner. This makes the principles of our proposed solution independent of specific implementations and environment assumptions (e.g., it does not depend on any specific distributed hash tables or specific network timing assumptions, etc). The contribution of this paper is therefore largely conceptual, as it focuses on defining the right architectural abstractions and on their orchestration, rather than on the actual mechanisms that implement each of its components.

** Most of the work was done while the author was visiting IRISA. This author is also partially supported by an IBM Faculty Partnership Award.

R. Meersman, Z. Tari (Eds.): CoopIS/DOA/ODBASE 2004, LNCS 3291, pp. 1445–1462, 2004.

1 Introduction

This paper considers the problem of providing a scalable self-managing infrastructure for executing distributed *atomic* operations on *persistent objects* in a dynamic and open distributed system. The main target of this infrastructure is to support operations between ad-hoc subsets of nodes taken from a much larger set Π of potential participants. For scalability reasons, the size and full composition of Π can not be known to any of its members at any given time, and may also change dynamically. Similarly, due to the scalability and self-management requirements, we rule-out solutions in which all interactions must be mitigated by the same relatively small set of nodes.

The immediate type of solution that comes to mind when considering these environments is to employ *distributed hash table* (DHT) based *peer-to-peer* systems such as Pastry [22], Tapestry [28], CAN [21], and Chord [25]. However, unlike most existing applications of peer-to-peer technologies, we are interested in providing stronger semantics between object operations. Also, in our model, although the entire set of nodes can be arbitrarily large, the number of nodes involved in a particular series of operations is relatively small, i.e., the same as the size of a (traditional) distributed system. Alternatively, one could consider solutions originating from the "classical" distributed computing model, such as *replicated state machine* [23] based on consensus [20,17] or group communication [5]. However, solutions from this domain do not scale well, and in some cases, rely on assumptions like having a fixed known size of the group of participants, or the existence of other services whose implementation in a self-managing manner is not trivial.

We claim that a good way to obtain large scale solutions that adhere to strong semantics is by providing a complete architecture that combines several peer-to-peer based services and known classical distributed computing solutions. This novel architecture helps designing protocols by reusing known building blocks, and also shows how these blocks can be combined to solve a particular problem.

Moreover, we advocate that factorizing the theoretically difficult part of these domains and encapsulating it in an appropriate "oracle" allows generic implementations of these solutions.

1.1 Contributions of This Work

We present an architecture for providing linearizable distributed operations on objects in scalable dynamic environments and provide an orchestrating algorithm that combines the components of the architecture into a complete solution. Our architecture identifies components that we recommend implementing using peer-to-peer technologies (we refer to these as *services*) and components that we recommend to implement using classical distributed computing techniques (we refer to these as *modules*). Basically, a service is supposed to be run on the whole system, while a module is to be instantiated on a small subset of nodes. The services include *event notification*, *conflict detection*, and *persistent storage*; the

modules include *consensus* and *group membership*. For each service and module, we provide a formal functional implementation-independent specification.

We then discuss the implementation of each of the components. For each component, we identify the aspects that can be implemented using very basic networking assumptions, namely an Internet like network, from the functionality that requires stronger assumptions. For the latter, we encapsulate the stronger assumptions within an *oracle*, for which we provide a precise functional implementation-independent definition. For example, in the case of consensus, it is known that a $\diamond\mathcal{S}$ *failure detector* and reliable point-to-point delivery capabilities are necessary and sufficient [7]. For the other services that we recommend building with peer-to-peer technologies, we define the corresponding oracles.

We would like to stress that we use the notion of an "oracle" for a subcomponent that requires additional environmental assumptions to be implemented. This allows us to only make very weak networking assumptions at the high level architectural description of the solution. All additional assumptions are deferred to the lowest level where they are needed, and are hidden at the architectural level behind a functional specification. This makes the solution "cleaner" and more generic.

In general, oracles are helpful in designing practical systems that can be reasoned about using formal methods. From a theoretical stand point, oracles allow the definition of "clean" models that enable formal reasoning, and in particular, designing protocols with formal proofs of correctness and the investigation of precise lower bounds and impossibility results. From a practical point of view, oracles allow abstracting away many low level assumptions, and provide a separation between the conceptual and the more technical part.

Notice that, for each of the oracles we define, there may be different implementations that require different low level environment assumptions. By specifying the functional property that we need, we can provide a generic solution that can then be translated to different environments with different assumptions and corresponding exact realizations.

Finally, we believe that our architectural oracle-based approach may be easily refined in order to address other distributed problems on arbitrary systems.

1.2 Problem Statement

We are interested in providing *linearizable*[1] *distributed multi-object operations*, which we will refer to in the rest of the paper as *transactions* [18].[2] More formally, from a programming model point of view, a *transaction* takes place between a set of nodes. A transaction is initiated when a node invokes an `initiate-transaction` method whose parameters include the transaction's

[1] An execution of a distributed system satisfies linearizability [15] if it could appear to an external observer as a sequence composed of the operations invoked by processes that respects objects specification and real-time precedence ordering on operations

[2] This is *not* data-base transactions, but rather a simple binding of multiple operations on objects in one atomic interaction.

context; this method returns a transaction identifier *tid* that is derived uniquely from its context. If two nodes initiate a transaction with the same context at the same time, they will obtain the same transaction identifier. After obtaining a transaction identifier *tid*, nodes can invoke a `join-transaction` method whose parameter is the transaction identifier *tid*; this method may return FAILURE if the transaction is no longer valid, or SUCCESS otherwise. Once a node joins a transaction *tid*, it can tag operations on objects it wishes to relate to this transaction with *tid*. Yet, we assume that each operation is tagged with at most one transaction identifier. Thus, we extend the classical notation of operations to `read`(tid, o, v) and `write`(tid, o, v) to express the fact that the operations are part of the transaction *tid*. Finally, at some point a node may invoke an `end-transaction` method with a proposed final status; this method returns with the final *status* of the transaction, which can be either ABORT, if the transaction failed, or COMMIT if the transaction succeeded. If a transaction ended with COMMIT, we say that it was *committed*; otherwise, we say that the transaction was *aborted*. Moreover, the transaction is said to be *pending* during the time interval between when the first node issues the `initiate-transaction` method and the time that the `end-transaction` method terminates at any node.

Two transactions are *concurrent* if the time intervals in which they are pending intersect. For a committed transaction *tid*, we define the following sets of objects: *read-set* is the set of objects read by `read` operations tagged with *tid*; *write-set* is the set of objects written by `write` operations tagged with *tid*; *transaction-operations* is the set of operations tagged with *tid*. These sets are empty for aborted transactions.

In the rest of the paper, we use the following notation to discuss the relation between transactions. Given two committed concurrent transactions $T1$ and $T2$, we say that $T1$ and $T2$ *conflict* if the *read-set* of either of them intersects with the *write-set* of the other, or if the *write-set*s of both transactions intersect. Also, in order to simplify the notation and formal definitions, we assume that a node can have at most one pending transaction at any given time. It is simple to extend all definitions and notations to remove this requirement in the natural way. The formal definition of the problem can be found in [3].

2 Overview of the Architecture

2.1 Major Components

The components we identify include:

Content-based notification service: This service distributes notifications to all interested parties, where the latter are determined according to the match between their known interests and the content of the notification [26,27].

Group membership module: This module allows nodes to join an abstract *group*; nodes that join the group become *members of the group*. Once a certain condition is satisfied (this condition is a parameter that is passed to this module), the membership service issues a *membership notification*

to all its members. Note that this is much weaker than a complete group communication toolkit, e.g., [10,13].

Distributed conflict detection service: This module generalizes the notion of a *lock* in traditional database systems. That is, it detects conflicting transactions in the system based on their proposed *read-sets* and *write-sets*, and when such transactions exist, it gives priority to one of them.

Consensus module: This module is a black-box consensus mechanism. That is, it allows a given set of nodes to reach agreement on a common value despite failures.

Persistent storage service: This service allows to store data in a persistent manner. The data is stored and accessed according to an identifier.

In the above, we distinguish between *services* and *modules*. In our terminology, a service is a global entity while a module is an entity that provides some properties only between an ad-hoc set of nodes. For example, the conflict detector service needs to detect conflicts between all concurrent transactions in the system. On the other hand, the membership module and consensus module can be instantiated only among the nodes that require them for a specific transaction.[3]

Also, we would like to point out that each of the above services and modules was (and still is) an active research topic by itself and there are known distributed solutions for each of these. Our main contributions are:

- Defining these components in a functional implementation-independent way.
- For each such component, identifying the "benign" part of it and the "oracle" part. For the "oracle" part, we provide a functional implementation-independent specification with clear interfaces.[4]
- Orchestrating all the components to work together as a solution that implements transactions semantics in a scalable self-managed manner.

2.2 Orchestrating the Components to Provide Transactions Semantics

In this section, we provide a generic orchestrating protocol for implementing linearizable distributed atomic transactions. The pseudo code of this protocol appears in Figures 1 and 2.

A transaction begins with the invocation of `initiate-transaction` for a given context. This function first computes a transaction identifier *tid* from the context of the transaction, then publishes this information using the content-based notification service. When a site receives a notification event from this service, it forwards it to the application that may decide to enter the transaction.

Then, all sites that want to participate start the actual transaction by a call to `participate-in-transaction`. This primitive first instantiates a membership object, then settles a group of participating sites, using the membership

[3] Of course, we do not rule out implementations in which the same instantiation of this module is utilized for more than one transaction, e.g., for performance reasons, but the functional properties of this module do not call for it.

[4] For consensus, the oracle approach was first introduced by Chandra and Toueg [8].

Variables:
notification – a handle for the notification service
conflict – a handle for the conflict detector service
storage – a handle for the storage service
consensus – a handle for the consensus module
membership – a handle for the membership module

Initialization:
notification = *naming*.`Bind`("Notification");
conflict = *naming*.`Bind`("Conflict Detector");
storage = *naming*.`Bind`("Persistent Storage")

Fig. 1. Main variables accessed by each node and their initialization

primitive `Join`. If successful, the join operation returns a list of members from which each site computes a read-set and a write-set. These are passed to the `Check-for-conflicts` method of the conflict detection service, which verifies if there exists another concurrent transaction that conflicts with the current one.

Following this, based on the actual membership list, the context and possibly local states, each node computes new values for their local state and the objects stored in the persistent storage, and their vote on whether the transaction should commit or abort. The consensus module is then used to ensure a unique decision on the outcome of the transaction. If the decision is to commit the transaction, then every object that must be updated is stored in persistent storage, new values of local variables are committed, locks on the conflict detector module are released, and the transaction is ended.

Notice that the above protocol does not explicitly handle recoveries of nodes that failed in the middle of a transaction. When considering crashes and recoveries, it is only interesting to examine cases in which after recovery, a node has lost all its memory that was not stored on stable storage. By assuming that nodes have access to local stable storage, it is possible to utilize any known recovery mechanism that is typically used for distributed transactions, e.g., [18]. For example, a node can write to its stable storage every time it starts participating in a transaction and the results of each such transactions once it is determined (committed or aborted). The node can also make updates to shadow copies of its variables, rather than to its real variables, and store their values on its local stable storage just before the `end-transaction` method. Then, during recovery, it can check if it has such stored variables. If yes, it checks the outcome of the transaction on the storage service. If the transaction committed, it commits the locally stored shadow variables. If the transaction is still pending, it tries to participate once more in the consensus module for deciding its fate. Otherwise, if the transaction aborted, it simply eliminates its shadow copies. Also, if no local persistent storage is available, a node can use the persistent storage service, but this bears the costs of such an interaction.

Later in this paper we propose a detailed description of the three services introduced above. For lack of space, we do not present the Consensus and Membership modules (their detailed description can be found in [3]).

```
Upon initiate-transaction(context) from application do
    tid := context2id(context);
    notification.Publish(context,tid);
    participate-in-transaction(context,tid)

Upon join-transaction(tid) from application do
        participate-in-transaction(context,tid)

Upon receiving a Notify(context,tid) from notification service do
    if I have not joined context already then
        deliver (context,tid) to the application
    endif

participate-in-transaction(context,tid)
    membership := new Membership(tid);
    membership.Join(status,members);
    if status = SUCCESS then
        compute read-set and write-set;
        if check-for-conflicts(tid,read-set,write-set) then
            perform local computation;
            compute my estimate for status;
            status := end-transaction(tid,status,members);
            if status = COMMIT then
                commit local variables;
                for each obj. o and val. v that should be stored in the pers. storage service do
                    storage.Write(tid,o,v)
                enddo;
            endif;
        endif;
        release-conflicts(tid);
    endif;
    return(status)

check-for-conflicts(tid,read-set,write-set)
    for i := 1 to threshold do
        if conflict.Check-conflicts(tid,read-set,write-set) then
            return SUCCESS
        endif
    endfor
    return FAIL

release-conflicts(tid);
    conflict.Release(tid)

end-transaction(tid,status,members)
    consensus := new Consensus(tid, members);
    return consensus.Decide(status)
```

Fig. 2. Skeleton for application level interaction with the transactions interface

3 The Services in Detail

3.1 Notification Service

The notification service allows to distribute anonymously and asynchronously notifications to all interested parties, where the latter are determined according to the match between their known interests and the content of the notification. Informally, the notification service is in charge of: (*i*) storing all subscriptions associated with the respective subscribers, (*ii*) receiving all relevant notifications from publishers, and (*iii*) dispatching all published notifications to the correct subscribers.

For the formal definitions, we adopt the model of [2,11,26,27]. In the terminology of this model, nodes that wish to receive *notifications* can *subscribe* with a given *filter*; when a node is no longer interested in these notifications, it can *unsubscribe*. A filter is a query expression composed by a set of constraints, joined by boolean operations. A specific filter issued by a specific `Subscribe` operation is also called a *subscription*, whereas the invoking node is called a *subscriber*. Additionally, nodes can *publish events* by invoking a `Publish` method. When a given filter f evaluates to TRUE for a given event e, we say that e *matches* f. We say that a node p_i is *notified* of an event e when the notification service invokes a `Notify` method on p_i. When a node issues a `Subscribe` method, it may take some time for the notification service to be aware of this operation, especially because of the delay encountered by all the entities composing the notification service until they receive the subscription request. Once this delay elapses, we say that the subscription is *stable*. We also denote T_{diff} the time that it takes the notification service to perform a diffusion of the information (i.e., the matching to compute the set of interested subscribers and the sending of the notification to them). Due to asynchrony, it is possible that this time is only known to "external observers", but not to the actual nodes of the system. The formal requirements from a notification system are:

Legality. If some node p_i is notified about some event e, then p_i previously subscribed a subscription with a filter f such that e matches f.

Validity. If some node p_i is notified with e, then there exists some node that previously published e.

Fairness. Every node may publish infinitely often.

Event Liveness. Node p_i is eventually notified with e if it subscribed a filter f such that e matches f and f is stable T_{diff} time units after e has been published.

To realize this model, we assume an interface consisting of the following methods:

`Notification`()
: A constructor that instantiates the service – executed once in the system.

`Subscribe`(**in** FILTER *filter*, **in** UPCALL *Notify*)
: The invoking node indicates that it is interested in being notified of events matching pattern *filter*. This node is called a subscriber.

`Unsubscribe`(**in** FILTER *filter*)
: The subscriber informs that it is no longer interested in receiving notifications of events matching pattern *filter*.

`Publish`(**in** EVENT *event*)
: The invoking node generates the event it wishes to publish. This node is called a publisher.

UPCALL `Notify`(**in** EVENT *event*,**in** FILTER *filter*)
: The subscriber receives a notification for the event *event* that was matched on filter *filter*.

As pointed out by Huang and Garcia-Molina [16], a publish/subscribe system is often implemented over a network of *brokers* that are responsible for routing

information or events between publishers and subscribers. Solving the notification problem comes down to (i) efficiently matching an event against a large number of subscriptions and (ii) efficiently arranging the network of brokers and propagating events within this network. At the same time, there are notification systems like [6] in which any node can also act as a broker. Thus, in this work, we consider being a broker as a role that can be played by any node, just like any other role (publisher and subscriber).

In order to capture these two goals of the event-notification system, we can identify the following *dissemination oracle*. The dissemination oracle supports the following method:

PID-LIST `Forward`(**in** PID *pid*, **in** EVENT *event*)

The `Forward` method returns the partial list of nodes to which the corresponding *event* should be propagated. In order to define the dissemination oracle, we introduce the following definition on the lists of nodes returned by invoking the `Forward` method successively for the same event, while avoiding invoking this method more than once from the same node. Formally:

Definition 1. [Pruned Recursive Invocation Chain:] *Given an event e and some node p_i, a* pruned recursive invocation chain *for e and p_i is defined as following: Let $pidlist_0$ be the list of nodes returned by invoking the `Forward`(p_i,e) method, and let $called_0 = \{p_i\}$. Next, for each $k > 0$, we choose any node $p_j \in pidlist_{k-1}$ and define $called_k$ to be $called_{k-1} \cup \{p_j\}$ and $pidlist_k$ to be $pidlist_{k-1} \cup \{$ the list of nodes returned from invoking `Forward`(p_j,e) $\} \setminus called_k$.*

Note that if the rate of changes in the system is not too fast, than due to the fact that at any given time the number of nodes is finite, all such invocation chains are finite. Otherwise, if there is no bound on the rate of changes in the system, an invocation chain could be infinite. Based on the above definition, the oracle is expected to provide the following property:

Eventual Full Coverage: For any published event e, the union of all partial lists of nodes returned by the `Forward` method during any pruned recursive invocation chain for e includes all the nodes that during this invocation chain have a stable subscription with a filter f such that e matches f (but may include other nodes as well; intuitively, such extra nodes serve as brokers for this event).

A simple implementation of a notification service based on this oracle can then be as follows: The interface to the notification system at each node p_i remembers all the filters to which p_i has subscribed (that is, the application at p_i issued a subscribe with no following unsubscribe). When a node p_i invokes the `Publish`(e) method, the notification service's interface first contacts the oracle to obtain a list of nodes to which the event should be forwarded; p_i then forwards the event to all these nodes. Every node p_j that receives such an event e for the first time checks all the local subscription filters. If e matches any of them, then e is notified locally. Also, p_j invokes (recursively) the `Forward` method to obtain

```
Local Variables:
    local_subscription – a table of (FILTER,UPCALL) tuples

Upon Subscribe(f, u) from the application do
    local_subscription.Add (f, u)
enddo

Upon Unsubscribe(f) from the application do
    local_subscription.Remove (f)
enddo

Upon Publish(e) from the application do
    nodes := dissemination.Forward(p_i,e);
    send (NOTIFY-MSG,e) to each node in nodes
enddo

Upon receiving a (NOTIFY-MSG,e) from p_j do
    forall s ∈ local_subscription such that s.filter matches e
            and e was not locally notified before do
        invoke s.Notify(e, f)
    enddo;
    nodes := dissemination.Forward(p_i,e);
    send (NOTIFY-MSG,e) to each node in nodes
enddo
```

Fig. 3. Implementing a notification service based on the dissemination oracle

the next list of nodes, and forwards e to them. This forwarding mechanism is pruned whenever the `Forward` method of the oracle returns an empty list. This protocol is summarized in Figure 3. Note that this description assumes that the oracle has an implicit knowledge of all subscriptions. This is done in order to maintain the generality of the presentation. In practice, the oracle may have additional methods that allow the service to explicitly notify it about active subscriptions.[5]

The bulk of the work on notification services [26,27] can be viewed as looking for techniques that implement the dissemination oracle. For example, it is easy to implement the dissemination oracle on top of an underlying overlay that guarantees that for any two nodes p_i and p_j such that p_j's subscription matches p_i publication, then there is a path between p_i and p_j.

In particular, the simplest implementation is to assume a reliable single broker, which remembers all subscriptions, as is done, e.g., in Siena [24] and Gryphon [14]. Whenever the oracle is consulted at a publishing node, the oracle returns the identifier of the unique broker. At the broker, the oracle returns all the subscribers whose filters match the event.

As a final example we consider SCRIBE [6]. SCRIBE is a topic based event notification system, and thus, each subscription and each event are identified by a given topic, which can be mapped into some hash value. For each topic, the system implicitly assigns a single broker, using its distributed hash table mechanism. Moreover, for efficiency, the system builds subscription trees for each

[5] This is somewhat similar to failure detectors [9], in which their definition does not specify how they learn about failures.

topic. Notifications are then pushed to the root of the tree, i.e., the responsible broker, and are then propagated along the reverse tree. Stating this approach in our dissemination terminology, the dissemination oracle simply tell each node about its father in the tree until the event reaches the root. After that, it returns each time the children in the reverse tree, until the event arrives to all leafs.

Many of these systems do not efficiently handle failures and recoveries, or very frequent connections and disconnections. Thus, they imply additional assumptions on the environment beyond the model of Section 1.2. Yet, each of these systems rely on somewhat different assumptions, promoting the use of the dissemination oracle as an oracle rather than as a service.

As a final comment, admittedly, it appears that the dissemination oracle we proposed is too strong, in the sense that it appears to solve too much of the essence of the problem. Thus, future research is needed in order to find a possibly different oracle that is weaker, but strong enough to solve the problem, and also generalizes existing solutions to implementing notification services.

3.2 Conflict Detection Service

The purpose of the conflict detection services is to help ensure that two concurrent transactions do not try to access the same objects in a conflicting manner (i.e., both try to write to the same object, or one tries to write to and the other tries to read from the same object). In many database systems, this is obtained by asking each transaction to explicitly lock objects it accesses using some single object locking mechanism. Yet, unless care is taken, locking objects one by one may cause deadlocks. As the applications we envision may involve different entities spread over a large area, it is not advisable to rely on having all of them conform to the same locking strategies. Moreover, from a performance viewpoint, it may be impossible to run deadlock detection and prevention protocols assuming independent object locking. Thus, the conflict detector imposes a programming model in which each transaction must declare at once all the objects it wishes to lock for reading (i.e, its *read-set*) and for writing (i.e, its *write-set*). The conflict detector then provides the equivalent of an atomic locking mechanism for all of these objects.

Formally, the conflict detector can be invoked by transactions with a `Check - for - conflicts` method that accepts two sets of objects, a *read-set* and a *write-set*, and with a `Release` method. The `Check-forconflicts` method then returns with a GRANTED or DENIED. When an invocation of the method `Check-for-conflicts` done by some transaction T with the sets *read-set* and *write-set* returns with GRANTED, we say that *the method mutually* (resp., *exclusively*) *grants the objects in read-set* (resp., *write-set*) to T. We say that T *releases* objects previously granted to it either when the `Release` method is invoked from T, or when all the nodes that participate in T crash or become permanently disconnected. Based on these definitions, we require the conflict detector to provide the following properties:

Validity-Safety: If a transaction T is granted an object o exclusively, then no other transaction is granted o exclusively or mutually until T releases

its locked objects. Additionally, if a transaction T is granted an object o mutually, then no other transaction is granted o exclusively until T releases its locked objects.

Liveness: Each invocation of any of the `Check-for-conflicts` and `Release` methods eventually returns.

Fairness: If a transaction T invokes the `Check-for-conflicts` method with the sets *read-set* and *write-set* infinitely often, then T is eventually granted mutually all objects in *read-set* and is granted exclusively all objects in *write-set*.

The interface to the conflict detector service includes the following methods:

`Conflict-detection`()
A constructor that instantiates the service – executed once in the system.

`Check-for-conflicts`(**in** TID *tid*, **in** OBJID-LIST *read-set*,
in OBJID-LIST *write-set*, **out** LOCKSTATUS *conflict*)
The invoking node tries to acquire locks for its transaction
whose identifier is *tid*
operating on the objects listed in *read-set* and *write-set*.
The returned value in *conflict* is GRANTED
if the locks are granted and DENIED otherwise.

`Release`(**in** ID *tid*)
Transaction *tid* releases all its locks.

In order to implement the conflict detector, we can use a *locking* oracle. The locking oracle allows locking and releasing locks for specific object identifiers. The locking oracle supports the following two methods:

LOCKSTATUS `Shared`(**in** OBJID *oid*, **in** TID *tid*)
This method allows a transaction to try setting itself as a shared lock holder for a given object.
It returns GRANTED or DENIED.

LOCKSTATUS `Exclusive`(**in** OBJID *oid*, **in** TID *tid*)
This method allows a transaction to try setting itself as the exclusive lock holder for a given object.
It returns GRANTED or DENIED.

`Release`(**in** OBJID *oid*, **in** TID *tid*)
This method releases the lock.

When an invocation `Shared`(o,T) (resp., `Exclusive`(o,T)) returns with GRANTED for some object o and transaction T, we say that *the method mutually* (resp., *exclusively*) *granted o to T*. After an object o is granted (exclusively or mutually) to a transaction T, we say that T *releases* o either when the `Release`(o,T) method is invoked, or when all the nodes that participate in T crash or become permanently disconnected. Given the above definition, the locking oracle is required to support the following semantics:

Lock-safety: If a transaction T is granted the exclusive lock on an object o, then no other transaction is granted the exclusive or shared lock on o until

T releases o. Additionally, if T is granted the shared lock on o, then no other transaction is granted the exclusive lock on o until T releases o.

Lock-liveness: Every invocation of any of the `Shared`(o,T), `Exclusive`(o,T), and `Release`(o,T) methods for any object o and transaction T eventually returns.

Lock-fairness: If the `Shared`(o,T) (resp., `Exclusive`(o,T)) method is invoked infinitely often for some object o and transaction T, then o is eventually mutually (resp., exclusively) granted for T.

The reason why the locking oracle is considered an "oracle" rather than a normal service is that in order to implement such a scalable system-wide locking mechanism, one must rely on additional assumptions beyond the model of Section 1.2. An implementation of the locking service appears in the full version of this paper [3].

3.3 Scalable Persistent Storage

The persistent storage service is responsible for implementing long-lived objects that support read and write operations, while providing *linearizable* read/write semantics [15] reliably for a long period of time. In order to implement this service, we define the following interface:

`Write`(**in** OBJID *oid*, **in** VALUE *value*)
Write *value* to the object *oid*.
`Read`(**in** OBJID *oid*, **out** VALUE *value*)
Read the object *oid*. The returned value is stored in *value*.

In order to meet the persistence requirement of the storage, we replicate each object and employ quorum based access patterns. Yet, for the required scalability and self-management goals, we combine the quorum pattern with peer-to-peer techniques. Moreover, unlike most peer-to-peer implementations of replicate storage, we are not interested in a specific distributed hash table, but rather are looking for a generic implementation that can be mapped to existing schemes. Thus, we introduce the k, t*-overlay* oracle that supports one method, namely `Route`(*pid*,*oid*), which accepts a node identifier *pid* and an object identifier *oid*, and returns a node identifier. Intuitively, a node that wishes to access an object, utilizes this oracle to implicitly route the requests to a quorum of nodes that replicate the corresponding object.

Before specifying the exact requirement from the `Route` method, we first introduce a few helpful definitions. For a given object identifier *oid*, we say that the concatenation of *oid* with some integer l, denoted *oid_l*, is a *derivative object identifier* of *oid*.

Definition 2. [**Transitive Invocation:**] *For a given object identifier oid, a sequence of invocations of the* `Route` *method in which all invocations are called with the same object identifier derivative oid_l and the node identifier used in the*

j^{th} *invocation is the one returned in the* $(j-1)^{th}$ *invocation is called* a transitive invocation of `Route`.

If there exists an integer l, which is some function of Π, such that in each transitive invocation of `Route` that consists of at least l invocations, the last two invocations return the same node identifier, then we say that *this sequence of invocations converges*; the node that is returned by these last two invocations is called the *converged node*. Based on these definitions, the k, t-overlay oracle must provide the following two properties:[6]

Convergence: For every object identifier derivative *oid_l* and every initial node identifier *pid*, every transitive invocation of the `Route` method converges.

k, t-Quorum: For any two nodes p_i and p_j and object identifier *oid*, let D_i^{oid} and D_j^{oid} be any two sets of size t of object identifier derivatives taken from $[oid\_1, \ldots, oid\_k]$. Let C_i (resp., C_j) be the set of converged nodes obtained by having p_i (resp., p_j) transitively invoke the `Route` method on each derivative in D_i^{oid} (resp., D_j^{oid}). Then $C_i \cap C_j \neq \emptyset$.

Load Balancing: For every two object identifier derivatives *oid_l* and *oid_k* and every initial node identifier *pid*, any pair of transitive invocations of the `Route` method converge to different nodes with high probability.

The convergence property above ensures that it is possible to use the overlay oracle for routing. Also, as in other quorum replication techniques, a node that invokes an operation should be able to communicate with only a small subset of all replicas. Yet, such a node should be sure that any other invocation of an operation on the same object will intersect with as least one of the replicas it has accessed. This is obtained by the k, t-quorum property above. Finally, to ensure fault-tolerance, it is important that routing requests with different derivatives of the same object will reach different nodes in the system. This is provided by the load balancing property above.

Given the k, t-overlay oracle, it is possible to implement read/write objects using the following quorum protocol, which is an adaptation of [4]. Each node maintains a time stamp of the last value it has for each object, which is a pair of an integer and node *id*; the latter used to break symmetry. In order to read an object, a node issues read messages to the k object identifier derivatives obtained by the natural numbers $1, \ldots, k$. These messages are then propagated by the `Route` method of the oracle until they reach their respective converged nodes. The latter send the current value they hold and the time stamp they have for this object. Once the originator of the requests gets t replies, it picks the one with the highest time stamp, and returns it to the application. In order to write a value to some object o, a node needs to ensure that the new value will be written with a time stamp that is larger than any value previously written to o. This is done by mimicking a read operation, and then sending the write to all derivatives with a time stamp that is larger than the maximum obtained.

[6] This may also be seen as an extension of the *quorum failure detector* oracle that was introduced in [12] in the context of classical distributed systems, and shown there to be the weakest needed to implement a register with any number of failures.

```
Upon Write(o, v) from the application do
    ∀l ∈ [1, ..., k] let p_l := overlay.Route(p_i,o_l);
    ∀l ∈ [1, ..., k] send (QUERY,o_l,p_i) to p_l;
    wait until (RESPONSE,o,-,-) is received from at least t nodes;
    let (ts, j) be the largest received in any (RESPONSE,o,-,(ts, j)) message;
    ∀l ∈ [1, ..., k] send (WRITE,o_l,v,(ts + 1, i)) to p_l;
    return
enddo

Upon Read(o) from the application do
    ∀l ∈ [1, ..., k] let p_l := overlay.Route(p_i,o_l);
    ∀l ∈ [1, ..., k] send (QUERY,o_l,p_i) to p_l;
    wait until (RESPONSE,o,-,-) is received from at least t nodes;
    let (ts, j) be the largest received in any (RESPONSE,o,v,(ts, j)) message
            and v the corresponding value in that message;
    return v
enddo

Upon receiving (QUERY,o_l,p_k) from the network do
    next := overlay.Route(p_i,o_l);
    if next = p_i then
        send (RESPONSE,o,v,(ts, j)) to p_k;
            % note that p_k is the originator of (QUERY,o_l,p_k). So p_i directly sends the response to p_k
            % (ts, j) is the timestamp that p_i associated with o
            % if o does not exist locally, its local timestamp is 0 and the value
            % is the default initial value
    else
        send (QUERY,o_l,p_k) to next;
    endif
enddo

Upon receiving (WRITE,o_l,v,(ts, j)) from the network do
    next := overlay.Route(p_i,o_l);
    if next = p_i then
        if (ts, j) is larger than the time stamp of p_i's copy of o then
            update the value of the local copy of o to v
            update the time stamp of the local copy of o to (ts, j)
            % if o does not exist locally, this initializes its copy
        endif
    else
        send (WRITE,o_l,v,ts) to next;
    endif
enddo
```

Fig. 4. Scalable Persistent Storage

The fact that nodes wait to hear from t converged nodes plus the intersection property of the oracle imply the correctness of this protocol. The pseudo code appears in Figure 4.

Clearly, each of the distributed hash tables based peer-to-peer systems, such as Pastry [22], Tapestry [28], CAN [21], and Chord [25], can be used to implement the k, t-overlay oracle with appropriate assumptions on the rates of crashes, recoveries, connections, and disconnections.

Our work in this section can be also compared to the work on scalable dynamic quorum systems proposed in [19] and [1]. The solution presented in [19]

uses a CAN like logical network, while the construction in [1] is designed on top of a dynamic de Bruijn logical network. The quorum systems implemented by these works are similar to what we need in the k, t-overlay oracle, but in these papers, it is tightly integrated with the register implementation and tailored to the specific DHT chosen.

4 Conclusions

We proposed an architecture that brings together several seemingly distinct research areas, namely distributed consensus, group membership, notification services (publish/subscribe), scalable conflict detection (or locking), and scalable persistent storage. All these components (modules and services) are orchestrated together in order to obtain (strong) consistency on an *a priori* unsafe system. The implementation of each such service and module was divided into a "benign" part and an "oracle" part; the implementation of the former relies on very basic networking assumptions (and the existence of an "oracle") while the latter requires some additional environmental assumptions.

As for oracles, our main goal is to introduce this concept both as a software engineering design pattern, and as a theoretical research topic. An important question in that respect is when should a component be considered an oracle, and when a "standard" component. We propose the following three rules of thumb: 1) An oracle must provide a *functionality* that is vital in order to solve a higher level problem. 2) The functionality of the oracle cannot be satisfied with the basic environment assumptions and must require additional assumptions, yet these assumptions are not needed for the rest of the system. 3) There are (or there are likely to be) more than one way of implementing the oracle, yet each of these methods depend on different sets of environmental assumptions (timing, communication patterns, rates of connections and disconnections, etc.). Having an appropriate oracle allows to abstract away specific environmental assumptions, yielding more robust and generic architectures and protocols. It also encourages factorizing the commonality, rather than concentrating on ad-hoc solutions. This approach is common in the area of distributed consensus and to some extent also group communication. In the other areas we consider, to the best of our knowledge, this is the first attempt to employ oracles. Improving our proposed oracles and finding the optimal ones in each case is a very interesting open research question.

References

1. I. Abraham and D.Malkhi. Probabilistic quorums for dynamic systems. In *Proc. of the 16th International Symposium on Distributed Computing (DISC'03) — LNCS 2848*, pages 60–74, 2003.
2. E. Anceaume, A. Datta, M. Gradinariu, and G. Simon. Publish/subscribe scheme for mobile networks. In *Proc. of the Annual ACM Workshop on Principles of Mobile Computing (POMC'02)*, pages 74–81, 2002.

3. E. Anceaume, R. Friedman, M. Gradinariu, and M. Roy. An architecture for dynamic scalable self-managed transactions. Technical Report 1610, IRISA, Inria, France, http://www.irisa.fr/bibli/publi/pi/2004/1610/1610.html, 2004.
4. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. In *Proc. of the 9th Annual ACM Symposium on Principles of Distributed Computing (PODC'90)*, pages 363–375, 1990.
5. K. Birman and R. Van Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, 1994.
6. M. Castro, P. Druschel, A-M Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *Journal on Selected Areas in Communications*, 20(8):100–111, 2002.
7. T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.
8. T. Chandra and S. Toueg. Time and message efficient reliable broadcasts. In *Proc. of the 4th Annual International Workshop on Distributed Algorithms (WDAG'90)*, pages 289–300. Springer-Verlag, 1990.
9. T. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. *Journal of the ACM*, 43(2):225–267, 1996.
10. G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(427–469), 2001.
11. A. Datta, M. Gradinariu, M. Raynal, and G. Simon. Anonymous publish/subscribe in p2p networks. In *Proc. of the 18th International Parallel and Distributed Processing Symposium (IPDPS'03)*. ACM-IEEE, 2003.
12. C. Delporte, H. Fauconnier, and R. Guerraoui. Shared memory versus message passing. Technical Report 200377, Distributed Programming Laboratory (LPD) Lausanne, 2003.
13. R. Friedman and R. van Renesse. Strong and weak virtual synchrony in horus. In *Proc. of the 15th Symposium on Reliable Distributed Systems (SRDS'95)*, pages 140–149, 1996.
14. Gryphon. Web site. http://www.research.ibm.com/gryphon/.
15. M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.
16. Y Huang and H. Garcia-Molina. Publish/subscribe in a mobile environement. *ACM Int. Workshop on Data Engineering for wireless and mobile access (MOBIDE'01)*, pages 27–34, 2001.
17. L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.
18. N. Lynch, M. Merrit, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.
19. M. Naor and U.Weider. Scalable and dynamic quorum systems. In *Proc. of the 22th Annual ACM Symposium on Principles of Distributed Computing (PODC'03)*, pages 114–122, 2003.
20. M. Pease, R. Shostak, and L. Lamport. Reaching agreement in presence of faults. *Journal of ACM*, 27(2):228–234, 1980.
21. S. Ratnasamy, M. Handley, P. Francis, and R. Karp. A scalable content-addressable network. In *Proc. of the ACM SIGCOMM*, pages 161–172, 2001.
22. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large scale peer-to-peer systems. *Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, pages 329–350, 2001.

23. F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.
24. SIENA. Web site. http://www.cs.colorado.edu/users/carzanig/siena.
25. I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of the ACM SIGCOMM*, pages 149–160, 2001.
26. A. Virgillito. *Publish/subscribe communication systems: from models to applications*. PhD thesis, University of Roma "La Sapineza", 2003.
27. A. Virgillito, R. Beraldi, and R. Baldoni. On event routing in content-based publish/subscribe through dynamic networks. In *Proc. of The 9th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS)*, pages 322–329, May 2003.
28. B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, 2001.