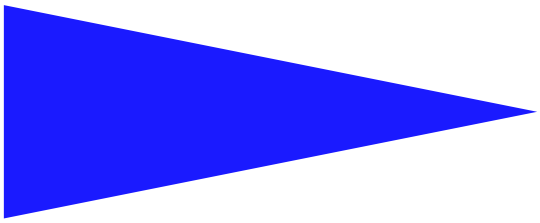
I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1610



AN ARCHITECTURE FOR DYNAMIC SCALABLE SELF-MANAGED DISTRIBUTED TRANSACTIONS

EMMANUELLE ANCEAUME ROY FRIEDMAN MARIA GRADINARIU MATTHIEU ROY

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

An Architecture for Dynamic Scalable Self-Managed Distributed Transactions

Emmanuelle Anceaume[**] Roy Friedman[*] Maria Gradinariu[**] Matthieu Roy[**]

Thème 1 — Réseaux et systèmes
Projet Adept

Publication interne n ° 1610 — March 2004 — 34 pages

Abstract: This paper presents a middleware architecture and a generic orchestrating protocol for implementing distributed atomic transactions 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 transaction 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 proposed architecture is also interesting in the sense that it brings together several seemingly distinct research areas, including distributed consensus, group membership, notification services (publish/subscribe), scalable conflict detection (or locking), and scalable persistent storage.

The 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.

Key-words: Dynamic Scalable Self-Managed Systems, Architecture, Oracles, Modules, Services, Distributed Transactions

(Résumé : tsvp)

[*] Computer Science Department, The Technion, Haifa 32000, Israel. roy@cs.Technion.AC.IL. Most of the work was done while the author was visiting IRISA. This author is also partially supported by an IBM Faculty Partnership Award.
[**] IRISA, Campus de Beaulieu, 35042 Rennes CEDEX, France. {anceaume,mgradina,mroy}@irisa.fr.

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes
Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Architecture pour la construction de services forte smantique dans les systmes grande chelle dynamiques - Cas d'tude : les transactions distribues

Résumé : Ce rapport prsente une architecture intergicielle pour la construction autonome de transactions distribues dans les rseaux grande chelle dynamiques. Un protocole gnrique orchestrant les modules de cette architecture est prsent. La solution que nous proposons est compltement distribue, adapte la mobilit des noeuds (les noeuds n'ont aucune connaissance de la taille du systme, ni a fortiori des noeuds qui le composent). Cette architecture est compose de deux modules et de trois services. Les modules sont instancis et excuts par les noeuds impliqus dans le contexte d'une transaction donne. Ceci permet leur implmentation dans un cadre purement "systmes distribus traditionnels". l'oppos, les services peuvent tre concerns par l'ensemble des noeuds du systme, et donc leur implmentation fait appel aux techniques utilises dans les systmes de pairs.

Mots clés : Systmes autonomes, extensibilit, oracles, modules, services, transactions distribues

1 Introduction

Self-management is one of the main requirements from future computing systems due to the following reasons: System administration costs are currently by far one of the major items in the total cost of ownership of computing systems. Additionally, in order for computing systems to scale to large sizes, as is potentially offered by the Internet, they must be able to adapt by themselves to dynamic changes that are inherent to such environments. These include, in particular, changes in the set of nodes that wish to participate in a computation as well as continuous failures and recoveries of nodes and even voluntary disconnections and reconnections of nodes. Moreover, large scale prohibits solutions that are based on a fixed, or even semi-fixed, set of servers that conduct the entire computation for all other nodes, as is common in the client-server model. It also calls for solutions that avoid, as much as possible, global knowledge of the entire set of nodes in the system.

In this work we are looking at the problem of providing a scalable self-managing infrastructure for executing *distributed atomic transactions* between ad-hoc subsets of nodes taken from a much larger set Π of potential participants. Moreover, for scalability reasons, the size and full composition of Π may not be known to any of its members at any given time, and can 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 [57], Tapestry [67], Viceroy [49], CAN [56], and Chord [62]. However, unlike most existing applications of peer-to-peer technologies, we are interested in providing strong semantics between transactions. Also, in our model, although the entire set of nodes can be arbitrarily large, each transaction involves only a relatively small subset of nodes.

Alternatively, one could consider solutions originating from the "classical" distributed computing model, such as *replicated state machine* [59] based on consensus [54, 44] or group communication [11]. However, solutions from this domain often 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.

Thus, we claim that a good way to solve the problem of scalable self-managed transactions is by providing a complete architecture that combines several peer-to-peer based services and known classical distributed computing solutions in an orchestrated manner. Moreover, we present such an architecture and discusses its benefits and limitations.

Comment: We are currently not aware of any established formal definition of peer-to-peer systems that distinguishes them from "classical" distributed computing. For this work, we will refer to solutions in which a node only has a partial knowledge of the entire set of nodes and in which nodes can communicate directly with only a small subset of the nodes in the system, but may have control of what this subset is, as *peer-to-peer*. On the other hand, we refer to the model in which the set of nodes is known and each node can communicate with each other, other than due to failures, as *classical distributed computing*.

1.1 Contributions of this Work

We start by formally defining the *augmented distributed atomic transaction* problem. This definition augments the distributed atomic transactions problem [48], which was originated in the area of distributed data-bases, to large scale dynamic environments such as the ones we are interested in. In particular, our definition does not assume a transaction manager, and does not require that all participants of a transaction will be known in advance.

We then present a middleware architecture for solving the augmented distributed atomic transaction problem 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*). 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.

Finally, we 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 [13]. On the other hand, for the services that we recommend building with peer-to-peer technologies, we define other corresponding oracles. For each of these services, we define the functional interface of its respective oracle, and discuss how it can be implemented using known peer-to-peer technology and additional assumptions on the environment (such as the rate of connections and disconnections, etc.).

We would like to stress that we use the notion of an "oracle" for a sub-component 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. There are many areas of computer science in which, largely speaking, there is a disconnection between theoreticians and practitioners. Part of this can be attributed to the difficulty of formulating clear rigorous models that are neither too restrictive nor too weak, yet allow precise reasoning and formal proofs. Practitioners, however, need to find working solutions and cope with the intricate aspects of the real world that are often hard to capture precisely in a general formal model. As been demonstrated in the distributed consensus and group communication problems, the use of oracles can bridge this gap. 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 stand point, oracles allow abstracting away many low level assumptions, thereby enabling generic solutions. It allows practitioners to design practical efficient solutions that are nevertheless based on sound and precise theoretical understanding. In that sense, oracles are extremely appealing.

Back to the consensus example mentioned before, it was shown that a failure detector of the class $\diamond\mathcal{S}$ is the weakest to solve Consensus in an asynchronous distributed system prone to crash failures [13] (we repeat the definition of $\diamond\mathcal{S}$ later in Section 4.1). Here, the functional property that is needed is some minimal level of *accuracy* in the ability to detect failures. The term "weakest" means that it is sufficient on one hand, and that any environment that can solve Consensus is also strong enough to implement $\diamond\mathcal{S}$. Yet, the important thing is that there can be different implementations of $\diamond\mathcal{S}$ that rely on completely distinct sets of assumptions. For example, the common way to implement $\diamond\mathcal{S}$ [16], and other failure detectors, is by exchanging *heartbeat* messages periodically and suspecting nodes from which such messages are not received after a timeout. Thus, this implementation relies on some timing assumptions, similar to the ones expressed in the *timed asynchronous* model [19]. Differently, it was shown in [51] that it is also possible to implement $\diamond\mathcal{S}$ using a *query-response* mechanism in an environment in which there are absolutely no timing guarantees, but the network ensures some minimal delivery ordering properties for responses (in particular, in the environment of [51], the network latencies may grow arbitrarily long, so no timeout based solution will work). Yet, a third option for implementing failure detectors, and in fact many of the other oracles we define in this paper, is to employ *wormholes*, as defined by Veríssimo [63]. The idea here is that the system is equipped with additional synchronous communication links, but these are used rarely, and only in order to ensure the oracles properties; all other communication travel on asynchronous links.

Similarly, 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.

As a final comment, this paper concentrates on architectural issues. The oracles we present are given as a proof of concept, aimed at promoting this approach. With the exception of consensus, which was studied thoroughly in that respect, the oracles for the other components are not claimed to be optimal. Finding such optimal oracles is an interesting open research question.

Paper roadmap: Section 2 presents the model assumptions and formal problem statement. Section 3 describes the architecture in general, while Section 4 details each of its components. Section 5 provides a few application examples, for validating our approach. We then conclude with a discussion in Section 6.

2 Computational Model and Problem Statement

2.1 Model

We assume a large finite, yet unbounded, set Π of *nodes*, also referred to as *processes* or *processors*, whose composition may change over time. Each node is associated with a unique identifier, and nodes can communicate with each other over a *network* providing a best-effort datagram service, similar to the Internet. That is, most messages are delivered, unless either the sender or receiver fails beforehand. Yet, there is no guarantee that a message from p to q will be delivered at all, and two consecutive messages sent from p to q may be received by q in the reverse order to the one they were sent in. Nodes also have a *local state*, which includes multiple *objects*. Each object has a *value* that can be changed as a result of a *local computation step* of that node.

For each node we define a *local history* to be a sequence of *events*, which consists of *connect*, *disconnect*, *sending* a message to other nodes, *receiving* a message from another node, taking computation steps, and possibly a *crash* event. If a crash event exists, it is always the last event in the local history. A collection of local histories, one for each node, is called an *execution*. In this work, we are only interested in *well formed executions*, in which if a message is received in the execution, then it was also sent during this execution. Finally, we refer to a well formed execution σ that can be extended to another well formed execution σ' as a sub-execution of σ'.

In a local history, the first event is always a *connect*; between a *disconnect* and the following *connect* event there can be no additional events; if a history includes a *disconnect* with no following *connect*, then there are no additional events in this local history. We say that a node is *connected* in the intervals of its history between a *connect* and the following *disconnect* (if exists); a node is *disconnected* in the intervals of its history between a *disconnect* and the following *connect* (if exists). If the last event in a local history of a node is a *disconnect*, then we say that this node became *permanently disconnected*. A node that crashes is sometimes also referred to as *faulty*, while a node that does not crash and does not become permanently disconnected is referred to as *correct*.

We assume that each transaction can be identified in an *associative* manner using the abstract notion of a *context*. Practically, the context can be a combination of the transaction's *topic*, e.g., selling a 1975 Volkswagen Kombi, some characterization of who might be interested in participation in it, e.g., car dealers in Brittany, etc. Yet, we assume that there is a way to generate a unique identifier for each context, e.g., using some hashing function.

2.2 Formal Problem Statement

In addition to the local state, we also assume a large universe of *abstract read/write objects* $\mathcal{O}$, where at any given time, each node $p_i \in \Pi$ may only be aware of a subset of $\mathcal{O}$, and nodes can only invoke operations on objects they are aware of. Each of these objects support both a *read* and a *write* operation. For objects that are part of the local state of a node, read and write operations are modeled completely as local computation steps. Yet, for objects that are accessed by more than one node, read and write operations can be thought of as an alias for sequences of message exchanges and local computation steps that together implement the abstract read/write semantics [7]. We denote a read operation on an object o that returns a value v by `read`(o, v); a write operation writing v to o is denoted by `write`(o, v). Also, we assume that each object has a predefined initial value. Moreover, given a sequence of operations S, we say that S is legal if every read operation in S returns the value written by the last preceding write operation to the same object in S (or the initial value if no such write operation exists).

As indicated in the Introduction, we are interested in providing *augmented distributed atomic transactions semantics*. 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 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 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 naturally extend the previous 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.

Definition 2.1 *For a given execution σ, we say that a sequence of operations S is a* legal transaction based serialization *of σ if*

- *S includes exactly the set of operations that belong to all transaction-operations sets in σ.*
- *All operations belonging to the same transaction T are ordered in S between the corresponding pair of* `join-transaction` *and* `end-transaction` *operations. Moreover, between these pair of* `join-transaction` *and* `end-transaction` *operations, there are no operations of other transactions in S.*
- *All operations of the same node appear in S in the same order as in σ.*

Definition 2.2 *We say that a given execution σ of the system preserves the augmented distributed atomic transaction semantics if it obeys the following requirements:*

Agreement: *All nodes that finish executing the* `end-transaction` *operation of the same transaction obtain the same* status *value.*

Liveness: *All correct nodes that invoke a* `join-transaction` *for a given transaction finish executing the corresponding* `end-transaction`*.*

Serializability: *There exists a legal transaction based serialization of σ.*

Note that many papers and books that discuss transactions specify the *Atomicity, Consistency, Isolation, Durability* (ACID) model as the requirements from a transaction system, e.g. [48]. We claim that the ACID model and Definition 2.2 are two complementing ways of presenting of the same model. The ACID definition is given in a very operational manner and focuses on the guarantees for each separate transaction. On the other hand, Definition 2.2 is more functional, and is expressed in terms of properties of the entire system execution. Yet, by the definition of a legal transaction based serialization S, it is clear that it implies the following: (i) Atomicity, due to the fact that all the operations of all committed transactions (and only those) will appear in S, (ii) Consistency and Isolation, due to the legality requirement and the fact that S serializes operations based on their transactions, and (iii) Durability, which follows from the legality of S and the fact that S includes all operations of all committed transactions in the system (e.g., even if a transaction B appears in S far away from another transaction A, then B still "sees" the effects of A).

3 Overview of the Architecture

3.1 Major Components

The components we identify include (see also Figure 1):

Content-based notification service: This service allows to distribute notifications to all interested parties, where the latter are determined according to the match between their known interests and the content of the notification [64, 65].

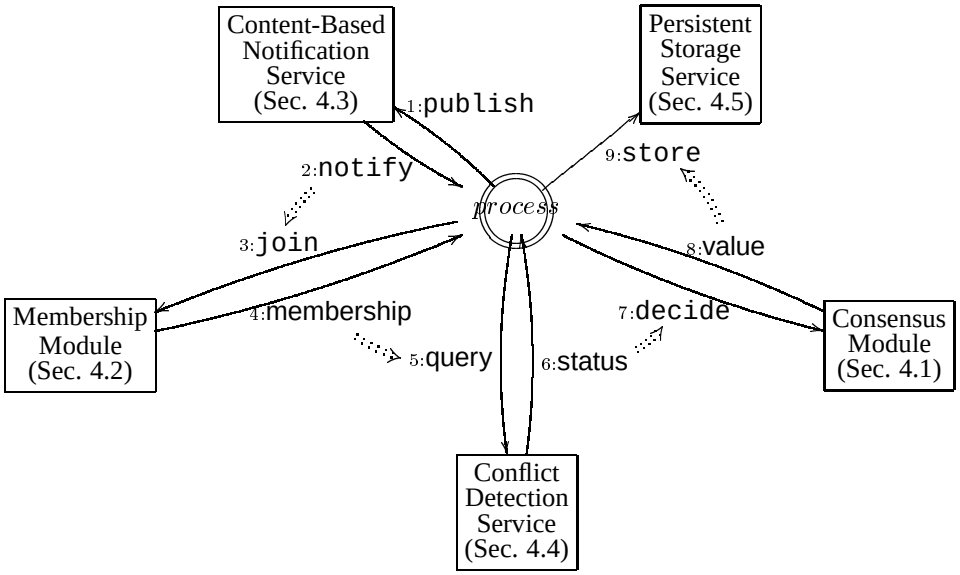


Figure 1: The Major Components and Their Orchestration

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., [17, 32].

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 allow 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.

Note that 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.[1]

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 building blocks 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.[2] We then provide a simple algorithm that implements the required functionality of the component based on the interaction with the "oracle". Finally, we describe possible implementations of the oracle using known technologies and the corresponding additional assumptions on the environment (that are necessary only for that "oracle").

[1] 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.

[2] For consensus, the oracle approach was first introduced by Chandra and Toueg [15].

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”)

Figure 2: Main variables accessed by each node and their initialization

- Orchestrating all the components to work together as a solution that implements transactions semantics in a scalable self-managed manner.

3.2 Orchestrating the Components to Provide Transactions Semantics

In this section, we provide a generic orchestrating protocols for implementing augmented distributed atomic transactions. The pseudo code of this protocol appears in Figures 2 and 3 (Figure 1 provides a schematic view of a node’s call chain to modules and services).

For a given site, a transaction begins with the invocation of `initiate-transaction` for a given context. This function first computes a transaction identifier *tid* from the context, 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 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 module, 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. 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., [48]. 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.

```
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 object o and value v that should be stored in the persistent 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)
```

Figure 3: Skeleton for application level interaction with the transactions interface

4 The Components in Detail

4.1 Consensus Module

Informally, the consensus module allows nodes of a given group to propose values and reach an agreement on a single value among the proposed ones. In the context of our work, this helps ensuring that all participants of a transaction agree on its outcome.

Formally, each node p_i *proposes* a value v_i, known as the *proposed value* of p_i. Unless p_i fails, or becomes permanently disconnected, p_i must eventually *decide* on some value u_i, known as the *decided value* of p_i. A solution to the consensus problem must provide the following three properties with respect to the proposed and decided values:

Validity: A decided value must be a value that has been proposed by some node.

Agreement: The decided values of all nodes that decide are the same.

Termination: Every correct node (recall, a node that neither crashes nor becomes permanently disconnected) must eventually decide on some value.

The consensus module exports the following interface:

`Consensus`(**in** GID *gid*, **in** MEMBERSHIP *membership*)
A constructor that instantiates the module.
The parameters are a group identifier *gid* and a membership list *membership*.
In particular, most implementations of consensus rely on knowing the membership list.

`Decide`(**in** VALUE *proposal*, **out** VALUE *decision*)
Try to decide on a proposed value *proposal*.
When this method terminates, it returns the decision value in *decision*.

It is well known that the consensus problem, as stated above, cannot be solved in purely asynchronous systems [27], known also as the FLP result. Thus, in order to solve it without weakening the definition of the problem, as is done, e.g., in [8, 9, 25, 30], one has to make additional assumptions on the environment. One option is to make some explicit synchronization assumptions, as in the *partially synchronous model* [19, 24, 26]. Yet, in this work we prefer to adopt the failure detectors approach [15]. That is, with the observation that the FLP impossibility result is based on the inability to detect failures in fully asynchronous systems, the system is enriched with an *unreliable failure detector* oracle. Such a failure detector can make some mistakes, but is good enough to solve the problem. Thus, rather than enriching the system with explicit timing assumptions, we enrich it with the functionality that is needed. Again, the benefit of this is that the protocols for solving consensus based on a failure detector oracle do not depend explicitly on timing assumptions, and are thus more robust. Moreover, weak failure detectors can be implemented in the partially synchronous model, e.g., [15]. Yet, recently, Mostefaoui and Raynal showed a query/response based implementation that depends only on the relative order of responses at each node, yet works even if network latencies grow arbitrarily large (so the partial synchrony assumptions never hold) [5, 51].

When one assumes that in each group at least half of its members will not crash or become permanently disconnected, it is sufficient to use a failure detector from the class $\diamond\mathcal{S}$. This class ensures that all crashed and permanently disconnected nodes are eventually detected, also known as *strong completeness*, while eventually, there exists one correct node that is not suspected by all other correct nodes, also known as *eventual weak accuracy*. That is, $\diamond\mathcal{S}$ can make many mistakes, but eventually there is at least one correct node that everyone trusts.

When the majority of correct nodes assumption does not hold, it was shown that the weakest failure detector is $\diamond\mathcal{S} \times \mathcal{P}^f$ [21].[3] The failure detector class $\mathcal{P}^f$ ensures strong completeness and that at most $n-f-1$ correct nodes are suspected at any time, where f is the maximal number of nodes that may fail or become permanently disconnected. A protocol that solves consensus for any $f < n$ using $\diamond\mathcal{S} \times \mathcal{P}^t$ can be found in [31].

Figure 4 provides a generic implementation for the consensus module, which utilizes any of the known consensus protocols for reaching the agreed decision. When a majority of correct nodes can be assumed, it employs any of the known $\diamond\mathcal{S}$-based consensus protocols, such as [16, 52, 58]. Otherwise, it uses the $\diamond\mathcal{S} \times \mathcal{P}^t$-based protocol of [31]. As all these protocols assume point-to-point reliable communication, i.e., that all messages sent from a correct node to another correct node are eventually delivered, the solution in Figure 4 must also assume such capability. Note that this can be implemented in the model of Section 2 without additional timing assumptions.

[3] In particular, $\diamond\mathcal{S} \times \mathcal{P}^t$ is strictly weaker than the classes $\mathcal{S}$ and $\mathcal{P}$ defined in [15].

Local variables:
gid – the unique group identifier
members – the membership list of the group

Upon `Decide`(v) from the application **do**
instantiate the consensus protocol for *gid* and *members* with v as my proposed value
If a majority of correct nodes is assumed, **then**
use any of the protocols in [16, 52, 58] with a $\diamond\mathcal{S}$ failure detector
Else
use the protocol of [31] with $\diamond\mathcal{S} \times \mathcal{P}^t$
endif
let u be the decided value by the above protocol;
return u
enddo

Figure 4: The Consensus Module Implementation

4.1.1 Consensus vs. Atomic Commit

Most existing transaction systems rely on the *atomic commit* specification rather than consensus in order to decide the fate of a transaction. Intuitively, the main difference between these problems is that in atomic commit each node has the right to veto the transaction. Specifically, if at least one node prefers to abort the transaction, than it must be aborted. More formally, in the *non blocking atomic commit* (NBAC) problem each node can propose YES or NO and must decide COMMIT or ABORT, while preserving the same Agreement and Termination properties as consensus plus the following validity property:

NBAC-Validity: A decided values is COMMIT or ABORT. Moreover:

NBAC-Justification: If a node decides COMMIT, all nodes have proposed YES.

NBAC-Obligation: If all nodes are correct and every node proposes YES, then the decision value is COMMIT.

Note that this definition allows to decide ABORT when all nodes propose YES only if at least one node has crashed or has become permanently disconnected. This is because if at least one node proposes NO, then everyone must decide ABORT. However, if someone has crashed, or becomes permanently disconnected, it may be impossible to know the value it initially proposed. Given the difficulty of accurately identifying crashed and permanently disconnected nodes, most systems solve a weaker problem called *weak non blocking atomic commit* (WNBAC), in which ABORT is allowed if at least one node is suspected to be failed or permanently disconnected.

In order to solve the WNBAC problem, we can use the same failure detectors as the ones used for consensus with very similar protocols [42]. On the other hand, for the strong version NBAC, it was shown that the weakest failure detector is $\diamond\mathcal{S}\times?\mathcal{P}$ when a majority of the nodes of a group are correct, and with $\diamond\mathcal{S}\times\mathcal{P}^t\times?\mathcal{P}$ otherwise [36] (among the *realistic* failure detectors, i.e., ones that cannot guess the future). The class of failure detectors $?\mathcal{P}$ provides the *anonymous accuracy* property; it returns TRUE if and only if a failure has occurred (but without giving any hints on who has failed).

So Which is Better: Consensus or NBAC? The reason why we chose to present the architecture with consensus rather than NBAC (or WNBAC) is that in the latter, a failure or permanent disconnection (or suspicion) of a node may cause the transaction to abort. In standard transaction systems, this may be acceptable, as this happens rarely. However, in large scale dynamic systems as we consider, such aborts may occur too often. As we wish to avoid aborts as much as possible, we have chosen to specify consensus. At the same time, it is easy to replace the consensus module with an NBAC or WNBAC module and obtain their semantics. The rest of the architecture and the orchestration protocol remain the same. Moreover, it might be worth investigating fine grained definitions that provide an intermediate step between consensus and NBAC (WNBAC) in the sense that a single failure does not necessarily impose ABORT, yet a certain threshold of NO votes will force an ABORT decision.

4.2 Membership Module

Informally, the functionality that we require from the membership module includes the following three properties:

- It should allow nodes that wish to join a group with some ID *gid* to do so.
- Once the set of nodes that joins the group matches some criterion, which is passed as a parameter to this module, it should send a final membership list to all the nodes that joined the group.
- After the membership list is sent, no new nodes should be able to join the group.
- All nodes that receive the membership list for the same group, receive the exact same list (same set of nodes).

Note that these are much weaker requirements than what is commonly required from a full fledged group communication toolkit [17, 32]. Formally, the membership problem we are interested in can be expressed as a one-shot distributed agreement problem as follows: Given the entire set of nodes Π, each node p_i may at some point try to *join* a group *gid* with a predicate P_i^{gid}. Without loss of generality, denote by $p_1^{gid}, \ldots, p_n^{gid}$ the set of nodes that wish to join the group *gid*. Each such node p_i^{gid} must eventually decide on a membership value m_i^{gid} such that m_i^{gid} can be either $-$ (a special value) or a set of node ids. We require the following

Agreement: For every two nodes p_i^{gid} and p_j^{gid} that decide, $m_i^{gid} = m_j^{gid}$.

Validity: $\forall i \in 1..n, m_i^{gid} \neq - \Rightarrow \big((\exists j : P_j^{gid}(m_i^{gid})) \ \wedge \ (\forall j : p_j \in m_i^{gid} \Rightarrow p_j$ tried to join *gid*$)\big)$.

Termination: Every correct node that tries to join a group *gid* eventually decides some value m_i^{gid}.

Fairness: If a correct node p_i tries to join infinite number of groups, then eventually there exists some group gid' such that $p_i \in m_i^{gid'}$.

The interface we define for this module consists of the following methods:

`Membership`(**in** GID *gid*, **in** CRITERIA *criteria*)
A constructor that instantiates the module for group *gid*.
The parameter *criteria* defines when to stop waiting for more members.
`Join`(**out** STATUS *status*, **out** MEMBERSHIP *membership-list*)
Indicate that the invoking member wishes to join the group.
The *status* output parameter is SUCCESS or FAILURE, where
FAILURE means that the request was unsuccessful, e.g., it arrived late.
The *membership-list* output parameter includes the list of members when *status* is SUCCESS;
it is undefined otherwise.

In order to realize this module, we must assume some oracle, as its solution requires stronger assumptions than our most basic assumptions of Section 2 [14]. Moreover, the main functionality that cannot be implemented in the basic model is ensuring that all nodes decide on the same membership list. Thus, we introduce the *primary view oracle*. This oracle exports two methods: `Compare&Swap` and `Verify`. The former method allows passing a proposed membership list and returns a non empty (but possibly different) membership list, while the latter does not accept any parameters and returns a (possibly empty) membership list. The primary view oracle ensures the following property with respect to the non-empty membership lists returned by these two methods:

Primary View: All non-empty membership lists returned by the `Compare&Swap` and `Verify` methods for the same group *gid* include the exact same set of node ids.

Using this oracle, we can provide the following implementation:

1. A node that wishes to join a group creates a local membership estimation that includes only itself.
2. Periodically, each node multicasts it estimated membership to all other nodes. It is important to note that this multicast does not need to be reliable and, in particular, does not need to provide any uniform delivery guarantee (see additional discussion below).

```
Local variables:
    gid – the unique group identifier
    members – the membership list of the group
    predicate – the predicate that the membership list must satisfy

Upon Join() from the application do
    members := {p_i};
    enddo
    % Notice that the control is not returned to the application here, i.e., it blocks

Periodically, if members ≠ ∅ do
    invoke mcast (MEMBERSHIP,gid,members);
    final := primary.Verify(gid);
    if final ≠ ∅ then
        invoke mcast(FINAL,gid,final);
        return final to the application
            endif
enddo

Upon receiving a message (MEMBERSHIP,gid,members_j) from the network do
    members := members ∪ members_j
    if predicate(members) = TRUE then
        final := primary.Compare&Swap(gid,members);
        invoke mcast(FINAL,gid,final);
        return final to the application
            endif

Upon receiving a message (FINAL,gid,final) from the network do
    return final to the application
        enddo
```

Figure 5: The Group Membership Module Implementation

3. Each time a node p_i receives a membership list from some other node, it unifies these lists. It then checks the termination predicate and if the predicate evaluates to TRUE, then p_i invokes the `Compare&Swap` method of the oracle, returns its result to the application, and terminates this instance of the module's execution.

4. Periodically, each node p_i for which the termination predicate never evaluated to TRUE invokes the `Verify` method of the oracle. If this method returns a non-empty list, then p_i returns this list to the application and terminates this instantiation of module execution.

A pseudocode version of this protocol appears in Figure 5. Note that the multicast service that we assume is implemented by IP multicast with the network assumptions of Section 2 (i.e., assuming that every message sent from a correct sender to a correct receiver is likely to be received). Another way of implementing such multicast when IP multicast does not exist is, for example, using a reflector service, such as the Ensemble gossip service [37].

The primary view oracle can be implemented in a straight forward manner in any system that provides a persistent Compare-and-Swap/Read-Modify-Write mechanism. A call to the Compare&Swap method of the oracle is implemented by trying to write to a register, and the write will succeed if the register was never written before; otherwise, the register is not modified. In both cases the value of the register is returned.

Let us also note that a Compare-and-Swap/Read-Modify-Write register is strong enough to allow solving consensus even in wait-free systems[4], meaning that it requires additional assumptions beyond the model of Section 2. Yet, different potential implementations imply different assumptions, justifying our designation of this functionality as an oracle.

[4] In a wait-free execution [38], no node can be prevented from terminating by undetected node disconnections/crashes or arbitrary variations in their speed

For example, one possibility of implementing a robust Read-Modify-Write register is to use replicated object techniques on a cluster of nodes, such as proposed in [2, 28, 29, 33, 41]. Such a clustered implementation could provide the Primary View oracle service to the entire system. Let us point out that all these solutions to replicated objects rely on the existence of some timing assumptions and/or failure detection capabilities between the nodes of the cluster. On the other hand, when the nodes of such a cluster are in physical proximity to each other, it is also possible to combine various hardware devices, such as SCSI locks and Storage Area Networks (SAN) fence operations, with shared RAID disks to obtain the required semantics, as can be found in offerings from companies like Polyserve Inc. and HP(Tandem) Inc., and research works like [18]. Here it is possible to trade timing assumptions on the network with the existence of special shared hardware. At any event, when considering such a solution for the oracle, other than this cluster, the rest of the system is self-managed and fully decentralized. An interesting topic for further research is to investigate enhancing peer-to-peer techniques that emulate Read-Write semantics to files [45] to implement Read-Modify-Write.

Clearly, it is also possible to implement the primary view oracle using a global total order multicast [10]. In this case, every node uses the first membership proposal it receives as its final membership list. However, the cost of a total ordering service in this case may be higher than other alternatives.

Finally, it is possible to add a gratuitous `Leave` method to the membership module, to allow nodes that change their mind before the membership was decided to leave the group. Such a `Leave` method should have a best-effort semantics, in the sense that if a node issues it after the final membership has already been computed, it must remain in the group.

4.3 Notification Service

The notification service allows to distribute anonymously, asynchronously, and in a loosely coupled manner 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:

- Storing all subscriptions associated with the respective subscribers.
- Receiving all relevant notifications from publishers.
- Dispatching all published notifications to the correct subscribers.

For the formal definitions, we adopt the model of [3, 20, 64, 65]. 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" of the system, but not to the actual nodes of the system.

We are now ready to formally define the requirements from a notification system. These 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.

In order 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 [40], 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 [12] 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).

Capturing 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 4.1 [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 is 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 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 6. Note that this description assumes that the oracle has an implicit knowledge of all subscriptions. This is done in order to

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 $s \in$ *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

Figure 6: Implementing a notification service based on the dissemination oracle

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]

For efficiency reasons, it may be worth limiting the number of notifications that each node may receive for the same event (at the service level). For this, it is possible to augment the dissemination oracle with the following efficiency property:

α, δ-**Redundancy:** For each event e, the number of times that a given node p_i appears in the returned list for different invocations of the `Forward`(-,e) method issued during the same time interval δ is less than α.

The bulk of the work on notification services [64, 65] can then 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 [61] and Gryphon [35]. 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.

A more complex, yet load balanced, mechanism was employed by Hermes [55]. In Hermes, each subscription is split into several pieces, each of which is potentially held by a different broker. When an event is published, the event is forwarded to all brokers that may hold subscriptions that could potentially match this event. These brokers then check their lists of filters, and forward events to all nodes they are aware of. The Hermes approach can be translated into an implementation of the dissemination oracle that does the following: At the publishing node, it returns the list of brokers that may have corresponding subscriptions pieces. At a broker b, the oracle returns the nodes for which the filter pieces held by b match the event.

Several topic and a content-based algorithms for peer-to-peer networks are presented in [3, 20]. In these works, the system is modeled as a self-organizing [4] multi-layer acyclic directed graph (DAG), such that each graph G^l represents the notification system for topic l. Each DAG contains two types of nodes, *sink* and *non-sink*. Only the

[5] This is somewhat similar to failure detectors [16], in which their definition does not specify how they learn about failures.

sink nodes are able to diffuse information. The non-sink nodes wait until they become sink nodes to disseminate information. The DAG orientation scheme ensures that all the non-sink nodes will eventually become sink nodes. Thus, the sink nodes forward messages that are generated locally or received from their neighbors. After sending their events, they re-orient their adjacent edges. In this approach, a simple implementation of the dissemination oracle is as follows: at the sink node, the oracle returns the list of interested nodes using the underlying oriented communication graph.

As a final example we consider SCRIBE [12]. 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. Thus, 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 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.

When considering all these systems, many of them do not handle well failures and recoveries, or very frequent connections and disconnections. Thus, they imply additional assumptions on the environment beyond the model of Section 2. Yet, each of these systems rely on somewhat different assumptions, promoting referring to 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.

4.4 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). Largely speaking, this ensures the Isolation property of the ACID model mentioned in Section 2. 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 `Checkfor-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*.

Thus, 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 2. As discussed below, it can be implemented in several ways, each of which depends on different assumptions. Notice also that the fairness requirement serves two purposes in this definition: First, it ensures fairness between transactions in the sense that a transaction cannot be denied of a lock it needs forever. Second, it also prevents trivial solutions in which locks are never granted to any transaction. By making some fairness assumptions about the environment, it is possible to weaken this requirement from the oracle, and replace it with an explicit non-triviality requirement.

Given that all objects that need locking by the same transaction are handed to the conflict detection service together, it is possible to implement the required conflict detector semantics with the following protocol, which is an adaptation of a known locking protocol taken from [48]: When the `Check-for-conflicts` method is invoked, the service first computes the *exclusive-objects* list to be the same as the *write-set* parameter, and the *shared-objects* list to be the *read-set* parameter excluding objects that also appear in *write-set*. The service then invokes the oracle's `Shared`(t,o) method for each object $o \in$ *shared-objects* and the `Exclusive`(t,o) method for each object $o \in$ *exclusive-objects* in an order that corresponds to the lexicographical order of the object identifiers. If the lock is denied for any such object, the service might try up to some threshold of times; if the service fails to obtain the lock afterwards, it releases all previously obtained locks and returns with a DENIED status. Otherwise, after obtaining all locks, the service returns with GRANTED. The `Release` method is realized by simply calling the `Unlock` method of the locking oracle. The

```
Variables:
    transactions-objects – a mapping from transaction identifiers to lists of objects

Upon Check-for-conflicts(tid, read-set, write-set) do
    exclusive-objects := write-set;
    shared-objects := read-set \ write-set;
    all-objects := write-set ∪ read-set;
    sort all-objects in lexicographical order;
    transactions-objects.Add(tid, all-objects);
    foreach o ∈ all-objects do
        if try-locking(o,tid) = DENIED then
            release-all-locks(tid);
            return DENIED
        endif
    enddo
enddo

Upon Release(tid) do
    release-all-locks(tid);
enddo

try-locking(o,tid,exclusive-objects,shared-objects)
    repeat at most th times
        if o ∈ exclusive-objects then
            if locking.Exclusive(o,tid) = GRANTED then
                return GRANTED
            endif
        else
            if locking.Shared(o,tid) = GRANTED then
                return GRANTED
            endif
        endif
    endrepeat
    return DENIED

release-all-locks(tid)
    objects := transactions-objects.Find(tid);
    reverse the order of objects;
    foreach o ∈ objects do
        locking.Release(o,tid)
    enddo
    transactions-objects.Remove(tid)
```

Figure 7: Conflict Detector Service

pseudocode for this implementation appears in Figure 7. The correctness and, in particular, the lack of deadlocks, result from the fact that objects are always obtained in lexicographical order [48].

When considering possible implementations of the locking oracle, it can be compared to the primary view oracle used in Section 4.2 to help determine a single group membership. The main difference between these two oracles is the fact that the primary view oracle supports a write-once semantics for each group, whereas this oracle is multiple-shot for each object. Hence, it may be easier to implement the primary view oracle. Another difference that stems from the previous one is the fact that the locking oracle can delay for a while a transaction that issues a `Lock` on an object that is already locked. This way, if the object is released shortly after, the oracle can return directly with the lock granted. In the primary view oracle, similar delaying of transactions is pointless.

The locking oracle can be implemented using a combination of Test-and-Set and Reset primitives. A node that wishes to lock an object, first tests the value of a binary register. When this value is 0, it modifies the register to 1 and uses the lock. Releasing a lock is done by reseting to 0 the register value. In order to implement Test-and-Set like primitives one can use similar techniques to the ones discussed for implementing a Read-Modify-Write register in Section 4.2.

The additional fairness requirement that the locking oracle must satisfy can be realized with an appropriate failure detection capability, similar to the perfect failure detector of [15]. That is, it depends on the ability to reliably detect when a node either crashes or becomes permanently disconnected. Alternatively, one can introduce the notion of *leases* [34]. That is, when a lock is granted to a given transaction, it is granted only for a given duration. If the transaction wishes to use the lock for a longer period, it must revalidate its ownership of the lock before it expires. Failure to revalidate a lock is implicitly translated into a release of that lock if another transaction is trying to obtain it. Implementing such a lease based mechanism requires strong timing assumptions, since it assumes that if a lock was revoked from a transaction due to a failure to revalidate the lock, then this transaction is aware of it. Yet, it is straight forward to present lease-based locks as oracles and modify the protocol of Figure 7 accordingly. Here the "oracle"-ish aspect of the lease-based locking mechanism is the assumption that either a correct node always manage to revalidate its locks in time, or that a node can know immediately when one of its locks has been implicitly released.

As far as we know the most efficient wait-free and lock-free solutions for transactions executions [50, 60] use a combination of the Multi-Word-Compare-and-Swap primitive with some "helping" mechanism. The combination of Test-and-Set, Reset and Leases that we describe above may be an interesting alternative to those implementation in terms of efficiency.

4.5 Scalable Persistent Storage

The persistent storage service is responsible for implementing long-lived objects that support read and write operations. More precisely, it must be able to store objects that support *linearizable* read/write semantics [39] reliably for a long period of time.

Consider the formal model of Section 2, and let $\mathcal{O}$ be a set of objects. For a given execution σ, we denote by $\sigma|\mathcal{O}$ the restriction of σ to operations on objects in $\mathcal{O}$. Also, let us note that each execution σ imposes a natural partial order on its operations: for every two operations op_1 and op_2 such that operation op_1 terminates before op_2 begins, op_1 is before op_2 in the partial order imposed by σ. For a given execution σ, we say that a sequence of operations S is a *linearization* of $\sigma|\mathcal{O}$ if S includes all the operation of $\sigma|\mathcal{O}$ ordered in an order that extends the partial order imposed on these operations by σ. We say that S is *legal* if every read operation in S reads the value of the last previous write operation to the same object in S (or the initial value of the object if there is no such write). Furthermore, we say that σ is $\mathcal{O}$-based linearizable if there exists a legal linearization of $\sigma|\mathcal{O}$. We define that a storage service storing a set of objects $\mathcal{O}$ implements *linearizability* if any execution σ of the system is $\mathcal{O}$-based linearizable.

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 4.2 [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.

In summary, given the k, t-overlay oracle, it is possible to implement read/write objects using the following quorum protocol, which is an adaptation of [6]. Each node maintains a timestamp 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 timestamp they have for this object. Once the originator of the requests gets t replies, it picks the one with the highest timestamp, 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 timestamp 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 timestamp that is larger than the maximum obtained. 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 pseudocode appears in Figure 8.

Clearly, each of the distributed hash tables based peer-to-peer systems, such as Pastry [57], Tapestry [67], Viceroy [49], CAN [56], and Chord [62], can be used to implement the k, t-overlay oracle with appropriate assumptions on the rates of crashes, recoveries, connections, and disconnections. Also, by adding timing assumptions to the model, it is possible to augment the definitions such that the intersection will only be ensured between transitive invocations of `Route` that occur within the maximal time it takes to perform a complete read and write operations. Once again, we would like to stress that the main contribution of this section is in factorizing this protocol out from existing works, thereby obtaining a generic solution that can be instantiated with different peer-to-peer systems and their corresponding environment assumptions.

[6] This may also be seen as an extension of the *quorum failure detector* oracle that was introduced in [22] 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**
    $\forall l \in [1, \ldots, k]$ let p_l := *overlay*.`Route`(p_i,o_l);
    $\forall l \in [1, \ldots, k]$ `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;
    $\forall l \in [1, \ldots, k]$ `send` (WRITE,o_l,v,$(ts + 1, i)$) to p_l;
    return
enddo

Upon `Read`(o) from the application **do**
    $\forall l \in [1, \ldots, k]$ let p_l := *overlay*.`Route`(p_i,o_l);
    $\forall l \in [1, \ldots, k]$ `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 timestamp of p_i's copy of o **then**
            update the value of the local copy of o to v
            update the timestamp 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

Figure 8: Scalable Persistent Storage

Our work in this section can be also compared to the work on scalable dynamic quorum systems proposed in [53] and [1]. The solution presented in [53] uses a CAN like logical network, while the construction in [1] is designed on top of a dynamic de Brujin 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.

As a final comment, *sequential consistency* is a slightly weaker consistency condition that is commonly implemented in distributed shared memory systems [43]. However, the problem with sequential consistency is that it is not *local* [39]. That is, a collection of sequentially consistent objects does not necessarily obey the specification of sequential consistency when considered as a whole. In fact, it was shown that most known formally specified consistency conditions that are stronger than or incomparable to PRAM [46] and weaker than linearizability are not local [66]. Clearly, being able to implement objects independently of each other and not having to synchronize operations on different objects is an important requirement for the scalability of the persistent storage service. At the same time, we would like the persistent storage service to support a meaningful and precise semantics that is strong enough for the role it plays in the orchestrating protocol of Figure 3. This led us to choose linearizability.

5 Example Applications

5.1 A Distributed Calendar

The distributed calendar application allows users to set meetings in a distributed manner. A constrained version of this application was developed as part of the Bayou project [23]. That is, in Bayou, the complete set of potential participants is known, and the system can rely on traditional data-bases, yet there is also support for temporarily disconnected nodes. In this work, we are interested is a slightly different formulation of this problem, in which the set of potential participants to each meeting is unknown a priori, and so are the possible meeting locations.

In considering our proposed architecture and the generic orchestration protocol of Figure 3, the calendar application can be realized as follows: When a node wishes to initiate a meeting, it publishes this fact using the notification service. All nodes that receive the notification and wish to participate in deciding on the date, time, and location of the meeting, can join the corresponding group using the membership service. Moreover, in this application, we can assume that the objects are the time-slots of meeting rooms and the time-slots of participants. Then, once the membership is fixed, they can share their availability with each other, and similarly, the availability of meeting rooms can be verified. Moreover, the use of the conflict detector service ensures that nodes can participate simultaneously in more than one decision attempt on a meeting, and that the same time-slot of the same room will not be reserved more than once.

Clearly, in this application it is also useful to store the decided meetings in the persistent store. This way, even nodes that did not participate in the decision on the date of a meeting can still learn about it and their users can attend the meeting if they wish to.

5.2 A Distributed Auction System

The idea of a distributed auction system is to generate a large scale dynamically adaptable auction system. This is somewhat similar to an eBay ® like system, but without a central authority. Also, we would like to support multi-party auctions and transactions. For example, in order to build a house, one needs to find a construction company, buy the different raw materials from several different vendors, hire electricians, plumbers, etc. Thus, the system we are proposing should allow auctioning all aspects of building a house simultaneously, and then run a single transaction during which all winning providers execute a mutual transaction in which they sign a joint mutual contract to build the house.

Our architecture allows solving this problem with a very slight extension of the orchestrating protocol. That is, each node that wishes to initiate such an activity can act as its own *auctioneer*. The auctioneer publishes the various auctions for the different parts of the task it wishes to complete using the event notification system. Every node that receives a corresponding notification and wishes to participate in the respective auction sends a bid to the auctioneer. Once the auctioneer decides on the winners of all auctions, it initiates a transaction among all the winners using the scheme described in Figure 3. In particular, the result of all auctions can be distributed using the notification service, and then the winners can join the corresponding group with the membership module. Also, the *condition* parameter

passed to the membership module in this case can be that all the winners of all relevant auctions (they are known at this point) join the group.

5.3 E-Flows

The idea of an e-flow is to allow users to combine a set of purchases from different entities into a single transaction. For example, a user may wish to take a vacation, which includes buying flight tickets from a certain airline company, a train ticket from the railway company, renting a car from one of the rental companies, and booking hotels in several hotels chains. As all these are part of the same vacation, the user may wish to bundle all of them into the same transaction. Similarly, companies may wish to conduct multi-party transactions in order to compose various raw material purchases, sub-contractors, and services, into a single transaction (or a contract).

This can be easily obtained with our proposed architecture in the following manner. The driving entity, be it the person looking for a vacation or the company that tries to manufacture some complex product, first looks at the various offerings and chooses the best one for him/herself in each category. Then, as in the auctions example, it notifies all the winners, who then join the transaction through the membership service, and then run the transaction to completion using the mechanism we described before.

6 Conclusions

This paper presented a middleware architecture that combines peer-to-peer based services with classical distributed computing based modules and an orchestrating protocol, which together provide scalable self-managed distributed transactions mechanism. Each of these services and modules was presented using a formal functional specification, and we also discussed how it can be implemented. Moreover, the implementation of each such service and module was divided into a “benign” part and an “oracle” part; the latter requires some additional environmental assumptions to be implemented while the former does not (except for the existence of the “oracle”).

We would like to emphasize that this paper does not pretend to invent any of the methods needed to implement each of the services and modules we discussed, as each of them is a large research topic by itself. Rather, our contribution is more conceptual. It consists of the architecture and the orchestrating mechanism. Our work brings together these various components that are typically studied independently of each other, and sometimes are even considered competing approaches. It shows how these components can be combined in a single middleware solution that enjoys the benefits of both worlds (peer-to-peer and classical distributed computing), and the benefits of all the schemes used to implement these components (publish/subscribe, group membership, consensus, scalable persistent storage, and scalable conflict detection). We hope that this will help viewing these areas as complementing, and will foster additional research into such hybrid solutions.

The architecture we presented here was tailored for the distributed transactions problem. However, we believe that its underlying principles are general, and could be applied, with minor modifications, to other problems that require strong semantics on one hand, and scalability and self-management on the other hand. We are currently investigating such additional problems.

With respect to the 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 *functional* behavior 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.

Finally, in this work we ignored security issues. Handling those, and in particular intrusion tolerance and Byzantine failures [44, 47] is left as an important open 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] Y. Amir. *Replication Using Group Communication Over a Partitioned Network*. PhD thesis, Institute of Computer Science, the Hebrew University of Jerusalem, 1995.

[3] 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.

[4] E. Anceaume, M. Gradinariu, and M. Roy. Self-organizing systems. case study: peer-to-peer networks. In *Proc. of the 16th International Symposium on Distributed Computing (DISC'03) — LNCS 2848*, 2003. Brief announcement.

[5] E. Anceaume, E. Mourgaya, and P. Raipin Parvdi. Converging toward decision conditions. In *Proc. of the 6th International Conference on Principles of Distributed Systems (OPODIS'02)*, pages 51–62, 2002.

[6] 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.

[7] H. Attiya and J. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

[8] Y. Aumann. Efficient Asynchronous Consensus with Weak Adversary Scheduler. In *Proc. of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC'97)*, pages 209–218, 1997.

[9] M. Ben-Or. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. In *Proc. of the 2nd Annual ACM Symposium on Principles of Distributed Computing (PODC'83)*, pages 27–30, 1983.

[10] K. Birman. *Building Secure and Reliable Network Applications*. Manning Publishing Company and Prentice Hall, 1996.

[11] K. Birman and R. Van Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, 1994.

[12] 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.

[13] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[14] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proc. of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, pages 322–330, 1996.

[15] 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.

[16] T. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. *Journal of the ACM*, 43(2):225–267, 1996.

[17] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(427–469), 2001.

[18] G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. In *Proc. of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC'02)*, pages 78–87, 2002.

[19] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, pages 642–657, 1999.

[20] 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.

[21] C. Delporte, H. Fauconnier, and R. Guerraoui. Failure detection lower bounds on registers and consensus. In *Proc. of the 15th International Symposium on Distributed Computing (DISC'02) — LNCS 2508*, pages 237–251, 2002.

[22] C. Delporte, H. Fauconnier, and R. Guerraoui. Shared memory versus message passing. Technical Report 200377, Distributed Programming Laboratory (LPD) Lausanne, 2003.

[23] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The bayou architecture: Support for data sharing among mobile users. In *Proc. of the IEEE Workshop on Mobile Computing Systems & Applications*, pages 2–7, 1994.

[24] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronization needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[25] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and H. Raymond Strong. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, 1986.

[26] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[27] M. Fischer, N. Lynch, and M. Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[28] R. Friedman. Using virtual synchrony to develop efficient fault tolerant distributed shared memories. Technical Report TR95–1506, Department of Computer Science, Cornell University, 1995.

[29] R. Friedman and E. Hadad. A group object adaptor-based approach to corba fault-tolerance. *IEEE Distributed Systems Online*, 2(7), 2001.

[30] R. Friedman, A. Mostéfaoui, S. Rajsbaum, and M. Raynal. Distributed agreement and its relation with error-correcting codes. In *Proc. of the 15th International Symposium on Distributed Computing (DISC'02) — LNCS 2508*, pages 63–87, 2002.

[31] R. Friedman, A. Mostefaoui, and M. Raynal. A weakest failure detector-based asynchronous consensus protocol for $f < n$. *Information Processing Letters*, 90(1):39–46, 2004.

[32] 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.

[33] R. Friedman and A. Vaysburd. Fast replicated state machines over partitionable networks. In *Proc. of the 16th Symposium on Reliable Distributed Systems (SRDS'97)*, pages 130–137, 1997.

[34] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. of the 12th ACM symposium on Operating systems principles (SOSP'89)*, pages 202–210. ACM Press, 1989.

[35] Gryphon. Web site. http://www.research.ibm.com/gryphon/.

[36] R. Guerraoui and P. Kouznetsov. On the weakest failure detector for non-blocking atomic commit. In *IFIP TCS*, pages 461–473, 2002.

[37] M. Hayden. The ensemble system. Technical Report TR98-1662, Department of Computer Science, Cornell University, 1998.

[38] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, 1991.

[39] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

[40] 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.

[41] I. Keidar. A highly available paradigm for consistent object replication. Master's thesis, Institute of Computer Science, the Hebrew University of Jerusalem, 1994.

[42] I. Keidar and D. Dolev. Increasing the resilience of atomic commit, at no additional cost. In *Proc. of ACM Symposium on Principles of Database Systems (PODS'95)*, pages 245–254, May 1995.

[43] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.

[44] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[45] K. Li. IVY: A shared virtual memory system for parallel computing. In *Proc. of the International Conference on Parallel Processing*, pages 94–101, 1988.

[46] R. Lipton and J. Sandberg. PRAM: A Scalable Shared Memory. Technical Report CS-TR-180-88, Computer Science Department, Princeton University, September 1988.

[47] N Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[48] N. Lynch, M. Merrit, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.

[49] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proc. of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC'02)*, pages 183–192, 2002.

[50] M. Moir. Transparent support for wait-free transactions. In *Workshop on Distributed Algorithms (WDAG'97)*, pages 305–319, 1997.

[51] A. Mostefaoui, E. Mourgaya, and M. Raynal. Asynchronous implementation of failure detection. *Proc. of the International Conference on Dependable Systems and Networks (DSN'03)*, pages 351–360, 2003.

[52] A. Mostéfaoui and M. Raynal. Solving consensus using chandra-toueg's unreliable failure detectors: a general quorum-based approach. In *Proc. 13th Int. Symposium on Distributed Computing (DISC'99)*, pages 49–63, 1999.

[53] 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.

[54] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in presence of faults. *Journal of ACM*, 27(2):228–234, 1980.

[55] P. Pietzuch and J. Bacon. Hermes: a distributed event-based middleware. In *Proc. of the 1rst International Workshop on Distributed Event-based Systems (DEBS'02)*, pages 611–618, 2002.

[56] S. Ratnasamy, M. Handley, P. Francis, and R. Karp. A scalable content-addressable network. In *Proc. of the ACM SIGCOMM*, pages 161–172, 2001.

[57] 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.

[58] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[59] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[60] N. Shavit and D. Touitou. Software transactional memory. In *Proc. of the 14st Annual ACM Symposium on Principles of Distributed Computing (PODC'95)*, pages 204–213, 1995.

[61] SIENA. Web site. http://www.cs.colorado.edu/users/carzanig/siena.

[62] 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.

[63] P. Veríssimo. Traveling through wormholes: Meeting the grand challenge of distributed systems. In *Proc. Int. Workshop on Future Directions in Distributed Computing (FuDiCo)*, pages 144–151, June 2002.

[64] A. Virgillito. *Publish/subscribe communication systems: from models to applications*. PhD thesis, University of Roma "La Sapineza", 2003.

[65] 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.

[66] R. Vitenberg and R. Friedman. On the locality of consistency conditions. In *Proc. of the 16th International Symposium on Distributed Computing (DISC'03) — LNCS 2848*, pages 92–105, 2003.

[67] 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.