Two Abstractions for Implementing Atomic Objects in Dynamic Systems

Roy Friedman[1], Michel Raynal[2], and Corentin Travers[2]

[1] Computer Science Department, Technion, Haifa 32000, Israel
[2] IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes, France
roy@cs.technion.ac.il,{raynal, ctravers}@irisa.fr

Abstract. Defining appropriate abstractions is one of the main challenges in computer science. This paper investigates two matching abstractions for implementing read/write objects in a dynamic server system prone to crash failures. The first abstraction concerns dynamic quorum systems. The second is a persistent reliable broadcast communication primitive. These two abstractions capture the essence of basic mechanisms allowing the implementation of atomic objects in a distributed system where servers can dynamically enter and leave the system (or crash). A read protocol and a write protocol based on these abstractions are described and proved correct. The properties defining these abstractions can be seen as requirements that are sufficient for implementing a dynamic storage service, while the feasibility conditions that are stated can be seen as necessary requirements. Instantiating the proposed abstractions in different contexts (e.g., settings defined by specific assumptions on failures, synchrony, message delays and processing times) provides as many system specific protocols.

Keywords: Atomic object, Communication primitive, Crash failures, Distributed system, Dynamic system, Quorum, Server, Shared memory.

1 Introduction

This paper is on the implementation of atomic read/write objects in a dynamic server system. More precisely, the general context that is considered is the following:

- There is an a priori infinite number of clients accessing shared objects. A client can sequentially issue read and write operations. It can also crash while executing an operation. A crash outside an operation is irrelevant.
- Each read or write operation on an object issued by a client is considered as an "atomic interaction" that accesses copies of the object. From an internal structure point of view, each operation follows the two phase pattern introduced in [5]. The first phase obtains control information, while the second phase ensures data persistence and consistency. This internal structure is unknown to the clients. From a client point of view, a read or write operation is a "primitive".
- Each object is supported by a set of servers. The server model is the infinite arrival model with finite concurrency [27]. This means that each run can have an infinite number of servers (i.e., an infinite number of servers can join and leave the system), but in each finite time interval there are finitely many servers. So intuitively, the only source of "infinitely" is the passage of time [1].

J.H. Anderson, G. Prencipe, and R. Wattenhofer (Eds.): OPODIS 2005, LNCS 3974, pp. 73–87, 2006.

This model is very general, and matches many types of long lived dynamic applications, as we elaborate in Section 6. In this paper, we are interested in implementing an atomic read/write object.

If servers join and leave the system arbitrarily fast, it is possible that no server remains long enough in the system for completing any read or write operations. So, implementing read and write operations requires some form of stability. This stability could be obtained by considering *duration assumptions* on the time a server process remains in the system, on message delays and on processing times. We consider here a more high level approach based only on the statement of *abstract properties* (in that sense our approach is similar to the failure detector approach introduced in [6]). Of course, implementing these properties can be done in dynamic systems satisfying some synchrony and durations assumptions (e.g., when there is a sliding time period of known and long enough duration during which some fixed and known number of non-faulty servers are continuously present).

So, instead of relying on specific low-level assumptions, the approach we propose to implement atomic shared objects in a dynamic distributed system is based on two complementary abstractions. The aim of these abstractions is to capture the relevant properties that facilitate the design of read and write protocols that focus on solving the problem rather than being overloaded with system specific implementation details. More precisely, we consider the following matching abstractions:

- The first defines quorums suited for read/write operations in a dynamic server system.[1] Each phase of an operation uses a particular type of quorum, and only some quorums have to intersect. More precisely, only the quorums of different types and belonging to consecutive operations have to intersect. Interestingly, the abstract properties defining these dynamic quorums can be interpreted as sufficient conditions when one wants to implement a dynamic reliable storage service. Feasibility conditions are also associated with these properties; those can be seen as necessary requirements for such implementations.
- The second abstraction, which we call *persistent reliable broadcast*, concerns communication. The primitives we propose allow an operation to broadcast a message uniformly to a sufficient subset of servers in a dynamic server model.

A read protocol and a corresponding write protocol that are based only on these abstractions are then presented and proved correct. Their correctness depends only on the properties of the abstractions. As those are defined as a set of abstract properties independent of a particular system or given technology, they can be implemented differently in different systems[2]. This modular approach favors the proof of the upper layer protocols, and cleanly separates between the properties we want to benefit from and their implementation [12].

[1] By definition, an object type that allows solving the Consensus problem despite process failures in otherwise asynchronous environment cannot be implemented purely by intersecting quorums. In particular, read-modify-write semantics is too strong to be supported by intersecting quorums without additional synchrony assumptions or failure detection capabilities.

[2] A trivial case being a static system with a majority of correct servers.

2 Application and System Model

An *application* is made up of clients processes that enter and leave the system (or crash). These processes can access read/write shared data objects. A client is not aware of other clients; it only knows that other clients can concurrently coexist. There is no global time notion accessible to the clients or the objects. This section defines the corresponding computation model.

To simplify the presentation, we assume the existence of a discrete global clock. This clock, whose domain is the set of integers denoted $\mathbb{N}$, is a fictional device that is known neither by the clients, nor by the objects.

2.1 Client Processes

From the application's point of view, the system consists of a possibly infinite set of sequential processes (called clients) that access a pool of shared read/write objects. The client process model we consider is sometimes called the *infinite arrival process with finite concurrency* [27]: the system has infinitely many processes, each run can have infinitely many clients, but in each finite time interval only finitely many processes can take steps [1].

Each client has an identity. These identities are such that no two clients have the same identity, and any two identities can be compared. A client knows its identity, but does not know the identities of the other clients. In the following we consider that a client identity is an integer, yet the client identities are not necessarily consecutive integers. The interested reader may refer to [1] for a protocol to "name the anonymous".

A client process can crash. In that case it stops its execution. A crashed process does not recover. Let us note that practically, this means that a process that recovers can re-enter the system as a new process, i.e., with a new identity. As a client process is not aware of the other clients, it has to terminate its operations (if it does not crash) whatever the behavior of the other clients. This means that the operations provided to the client processes have to be *wait-free* [20] (with respect to other clients).

2.2 Shared Objects

Each object x of the shared memory can be accessed by two operations denoted READ (x) and WRITE (x, v). They allow the invoking process to obtain the value of x, or define the new value v of x, respectively. Each object x is *atomic*. This means that, from an external observer point of view, all the operations accessing x can be totally ordered in such a way that (1) this order respects their real-time occurrence order, and (2) each read obtains the value written by the last write that precedes it in this total order [23]. Atomicity is a fundamental concept as it allows us to reason sequentially despite concurrency.

Let us note that in the context of concurrent objects, i.e., objects that can be concurrently accessed by several processes, the *atomicity* concept has initially been formalized and investigated for read/write shared objects [23]. It has then been extended under the name *linearizability* to any object that has a sequential specification [21].

Interestingly, it has been shown that a run that satisfies the atomicity (linearizability) consistency criterion with respect to each shared object considered separately, also satisfies this criterion when we consider the whole set of atomic objects as a single (bigger) variable [21]. This property is called *locality*. Thus, atomic consistency is *local*. However, sequential consistency and causal consistency are not [21]. That is, merging a protocol providing sequential consistency on a single object x with a protocol providing sequential consistency on another object y does not provide a protocol providing sequential consistency on the composite object $X = [x, y]$.[3]

Locality is significant for both theory and practice. From a theoretical point of view, it allows us to reason sequentially on the combined set of all objects as if it was a single object. From an implementation and software engineering point of view, this property enables scalable composable realizations. That is, as soon as we have a protocol implementing atomic consistency for one object, we can run multiple independent instantiations of this protocol, one for each object, and the entire system will behave correctly without any additional control or synchronization.

2.3 Shared Memory: A Set of Servers

We consider a shared memory service consisting of read/write objects that are implemented on top of a distributed message-passing system made up of a set of server processes, denoted $s_1, s_2, ...$ As indicated in the introduction, this system may have an infinite number of servers. Yet (as for clients), in each finite time interval there is only a finite number of servers (*infinite arrival model with finite concurrency*). A server s_j can enter the shared memory service (event $init_j$). It can later crash (event $fail_j$) or leave the system (event $leave_j$). As we noted earlier, this means that a process that crashed or left the system can re-enter the system, each time with a new identity. Each object is implemented by a subset of servers. Practically, this allows us to assume that the subset of servers implementing a single object at any given finite time interval is reasonably small, even though the system as a whole might include a huge number of servers. Due to the locality property of *atomicity* (recall the discussion in Section 2.2), in the rest of the paper we consider a single object x without losing generality.

Let $up(t)$ denote the set of servers (implementing object x) that joined the system before time t and have neither crashed nor left at t. We assume that $\forall t : \ up(t) \neq \emptyset$. This is a *feasibility* condition necessary to obtain *live* quorums, i.e., quorums that can prevent from definitive blocking the read and write operations that use them.

2.4 Operations as Intervals

As we have seen, an application process can only invoke a read or a write operation on a shared object. These operations are abstract for it in the sense that it can use them as primitives but it does not know how these primitives and the atomic objects are implemented at the underlying level.

Let us consider the ath READ () or WRITE () operation invoked by the same client process p_i. The beginning of the execution of that operation at the client defines an

[3] The bounds of the locality property with respect to various consistency criteria have been investigated in [33].

event that we denote $start_i^a$. Similarly, its termination at the client defines an event that we denote end_i^a. The crash of a client p_i while it is executing a read or write operation defines an event that we denote $crash_i$ (let us note that the crash of p_i outside an operation is irrelevant). On the application side, these are the only relevant events.

The invocations of read and write operations by a client p_i defines its local history. The subsequence of events between $start_i^a$ and end_i^a (or $crash_i$) defines what we call the *interval* I_i^a [18]. Let us stress that an interval is defined with respect to events at the client only, regardless of any events and operations taken by the servers or other clients. An execution of a set of processes sharing a set of atomic objects can be represented by a history h that is the sequence of events issued by these processes (if two or more events are "simultaneous", they can be arbitrarily ordered [22]).

Interestingly, the history h defines a natural partial order on the intervals. $I_i^a \rightarrow_h I_j^b$ (*precedes*) if end_i^a (or $crash_i$) appears in h before $start_j^b$. I_i^a is an *immediate predecessor* of I_j^b if $I_i^a \rightarrow_h I_j^b$ and there is no interval I such that $I_i^a \rightarrow_h I$ and $I \rightarrow_h I_j^b$. Finally, $im\_pred(I1, I2)$ is a predicate that is true if and only if $I1$ is an immediate predecessor of $I2$.

Let I be an interval whose start and end events occur at time t_b^I and t_e^I, respectively (if there is no end event for I, let $t_e^I = +\infty$). The following set of servers is associated with each interval I:

$$STABLE(I) = \{s \mid \exists t \in [t_b^I, t_e^I] : \ \forall t' : \ t \leq t' \leq t_e^I : \ \ s \in up(t') \ \}.$$

Another feasibility condition necessary to obtain live quorums is to have, for any interval I, $STABLE(I) \neq \emptyset$.

3 A Dynamic Read/Write Quorum Abstraction

3.1 Quorum Oracle

A quorum oracle is a device that provides the processes with a single primitive, namely a query. Moreover, we consider here that such a query can only be issued at a client due to a READ () or a WRITE () operation on a shared object inside the corresponding interval. Each query returns a set of servers. To be meaningful, the sets of servers returned by the queries have to satisfy some properties. A given set of such properties defines the type of the corresponding quorum oracle.

3.2 Dynamic Read/Write Quorums

We are now in order to define a class of quorum oracles that can be used to implement an atomic object in a dynamic server system. This class, denoted $\mathcal{RW}_{dyn}$, allows a process to issue two types of queries. As elaborated below, the goal of the first type is to obtain a "consistent" timestamp (associated with the value read or written), so we denote it CD (for control data). The second is to ensure that "enough" servers will have an up to date copy of the last value of the object, so we denote it VAL. $\mathcal{RW}_{dyn}$ is defined by the following properties:

– Progress property.
Let $Q(t)$ be the quorum obtained by a query issued at time t during an interval I (whatever the type CD or VAL of the query).

$$\exists t \in [t_b^I, t_e^I] : \quad \forall t' : t \leq t' \leq t_e^I : \; Q(t') \subseteq STABLE(I).$$

This property states that, by repeatedly querying its quorum oracle, an operation (that does not crash) eventually obtains a quorum of servers that have joined the system and have neither crashed nor left the system.
– Typed Bounded Lifetime Intersection property.
This property involves the two types of queries and their associated intervals. It states that the quorums returned by two such queries have a non empty intersection only if these queries (1) have different types and (2) belong to consecutive intervals. Let Q_{cd} (resp., Q_{val}) denote both the quorum returned by a query whose type is CD (resp., VAL), and the corresponding query event. Let $I1$ and $I2$ be the intervals associated with these queries. We have:

$$[(Q_{val} \in I1) \; \wedge \; (Q_{cd} \in I2) \; \wedge \; im\_pred(I1, I2)] \Rightarrow Q_{val} \cap Q_{cd} \neq \emptyset.$$

3.3 Related Quorum Systems

When comparing $\mathcal{RW}_{dyn}$ with traditional quorum systems [14, 15, 32], a noteworthy difference lies in the limited period during which quorums (of different types) have to intersect[4]. Interestingly, this intersection requirement allows all the servers that are alive and participate in a quorum at a given time to later crash or leave the system. In contrast, the *quorum failure detectors* introduced in [7, 8] require that all quorums will intersect in at least one process that never crashes. The generalization of quorum failure detectors in [11] only requires intersections between concurrent and immediately consecutive quorums, but does not allow all the servers that are alive at some point to later crash.

Herlihy's work describe a scheme that allows processes to switch between quorums, e.g., due to partitions [19]. The work of Herlihy concentrates on the mechanisms for performing such transformations and assumes a finite set of servers. Our work, on the other hand, concentrates on the formal framework and definitions of quorums in a dynamic system. In our approach, the change in the set of servers is inherently decided by the environment and cannot be controlled by the processes.

The class $\mathcal{RW}_{dyn}$ differs also from the quorums as defined in the seminal work on RAMBO [24]. RAMBO is a reconfigurable atomic memory service for dynamic networks. A key notion in RAMBO is the concept of *configuration* that is a set of members plus sets of read quorums and write quorums. RAMBO requires that any read quorum and any write quorum of the same configuration do intersect. Moreover, this intersection requirement is independent of the actual pattern of read and write operations (in

[4] One server that has the latest value of the object (it appears in the Q_{val} quorum) has to survive until the next operation (that obtains the quorum Q_{cd}), so that the previous intersection property can be satisfied.

our case, only consecutive operations require typed quorums to intersect). Thus, our intersection requirement may allow for more continuous evolution of the system.

The notion of a Byzantine quorum system, i.e., one that is resilient to Byzantine failures, was introduced in [25]. An extension that allows dynamically modifying the resilience threshold, yet with a constant set of servers, was introduced in [3]. The work of [29] describes a method that allows to dynamically change the set of servers by running Byzantine consensus to decide on the next configuration of the system, and thus can be thought as a kind of a Byzantine RAMBO like system. A somewhat similar approach of switching quorum systems using views was taken in [26]. However, in [26], a view change is performed by having an external entity notify a quorum of the current view to stop accepting requests in that view, and then notifying all members of the new view of its existence and initial state.

Finally, the idea of implementing a distributed shared memory in a dynamic system based on a group communication system was introduced in [10]. Rather than using quorums, that work relies on the virtual synchrony and total ordering mechanisms of the underlying group communication toolkit to obtain total ordering of operations and state continuity.

3.4 The Static Case

The static case is when the server system is statically defined with $m = 2f + 1$ servers, and up to f of them can crash. Moreover, the bound f is known by the processes. In this system, the classical quorum definition as sets of $f + 1$ servers trivially satisfies the two requirements of the previous definition. It is important to notice that if, incidentally, a run has more than $f + 1$ servers that crash, the Progress property can no longer be ensured, and operations based on such quorums can block forever. This means that, be the system dynamic or static, there are assumptions for the operations to terminate correctly. Here the implicit assumption is that "no more than f servers crash" (even when this assumption is embedded into the model, it may or may not be satisfied during a particular run).

4 A Communication Abstraction

In addition to the classical one-to-one reliable *send* and *receive* communication primitives, the underlying system offers two communication primitives prst_broadcast() and prst_deliver(). The first is to allow a read or a write operation to send a message to the set of servers. The second allows a server s to be delivered the corresponding message.

These primitives assume that each message m has a type $type(m)$ and a sequence number $sn(m)$. When a process executes prst_broadcast(m) (resp., prst_deliver()), we say that it "broadcasts" (resp., "delivers") m. The *persistent reliable broadcast* communication abstraction is defined by the following properties:

- Validity. If a message m is delivered by a server, it has been broadcast as part of the execution of a read or a write operation.
- Integrity. A message m is delivered at most once by each server.

- Server/server Termination. If a message m is broadcast during an interval I and is delivered by a server, then any server $s \in STABLE(I)$ eventually delivers a message m' such that $type(m) = type(m')$ and $sn(m') \geq sn(m)$.[5]
- Client/server Termination. If the client process does not crash while it is executing the read or write operation defining the interval I that gave rise to the broadcast of m, the message m is delivered eventually by at least one server.

The validity and integrity properties are *safety* properties. The first states that no spurious message is created, while the second states that no message is duplicated. The two other properties address the *liveness* of message deliveries. The client/server termination property states that if the application process that executes a read or write operation does not crash while it is executing that operation, each message it broadcasts (during that operation) is not lost in the sense that it is eventually delivered by at least one server. Due to asynchrony and the fact that servers can crash, or dynamically join/leave the system, it is not possible to require that all the servers that are active when a message m is broadcast will deliver the message. Hence the rationale for the server/server termination property that states that if a message is delivered by a server, then all the servers that have entered or will enter the system and neither leave it nor crash by the end of the operation (the servers defining the set denoted $STABLE(I)$), will deliver this message or a message of the same type sent later[6].

When all the messages have different types, the type notion disappears and sequence numbers become useless. If additionally the number of servers is statically defined, and all the events define a single interval [18], the primitives prst_broadcast() and prst_deliver() then boil down to the classical uniform reliable broadcast primitives [17].

An implementation of the *persistent reliable broadcast* abstraction can be done according to the following lines. When a server receives a message m, the server first forwards m to all the other processes, and only then delivers the message to itself (the way message forwarding is ensured depends on the underlying overlay network and the associated routing [28, 30, 31] – see also discussion in Section 6). Moreover, a new server that joins the system has first to broadcast (using the underlying routing) an inquiry message to the servers currently present in the system. When a server receives such a message, the server sends back its state and, for each message type, the sequence number of the last message it has delivered.

5 An Atomic Object Service

Assuming the previous dynamic quorum and persistent reliable broadcast abstractions, this section presents and proves correct a simple and general protocol implementing read and write operations suited to dynamic server systems.

[5] Notice that unlike uniform delivery, here the message m' that is eventually delivered by the servers in $STABLE(I)$ can be different from m, as long as the types of m and m' is the same and $sn(m') \geq sn(m)$.

[6] The underlying idea is here the following. A message m' that is causally affected by a message m (hence $sn(m') > sn(m)$) "includes" m from a causality point of view, and consequently the delivery of m' implicitly contains the delivery of m.

5.1 Structure of the Implementation

Each client p_i has a local variable sn_i that it uses to generate local sequence numbers. This allows p_i to give a unique identity to each read and write operation it invokes. In the following we consider an application process p_i and a read/write object x.

As a side comment, let us note that in some systems, an application process communicates only with a proxy, and several processes can share the same proxy. The proxy plays the role of the process with respect to the server processes. As an example, we have the following correspondence with transaction systems: transaction $\leftrightarrow$ operations, transaction manager $\leftrightarrow$ proxy, data managers $\leftrightarrow$ servers, and data $\leftrightarrow$ shared object copy. Here we could envisage a similar architecture, but as our focus is on atomic consistency, we do not detail the architectural issues of the whole system. Intuitively, the reader can think that the sequence numbers may be managed by the proxies and not by the processes themselves (as done, e.g., in [9]).

Back to our model, the protocol uses a classical timestamping mechanism [22]. It associates a timestamp ts, which consists of a pair made up of an integer denoted $ts.clock$ plus a process id denoted $ts.proc$, with each value that has been successfully written. Using lexicographic ordering, this allows us to obtain a total order on all the values that have been written. This total order is used to enforce atomic consistency. This basic principle is used in most atomic consistency protocols we are aware of.

The protocols implementing write and read operations are described in Figure 1 and Figure 2, respectively. They are based on the principles used in [5], namely, they are two-phase protocols. We first describe the write protocol, and then the read protocol.

5.2 Implementing a WRITE (x, v) Operation

When an application process p_i wants to write a new value, its first phase consists of defining a correct timestamp for the value v. The second phase is for p_i to ensure that the new pair (value, timestamp) is known by enough servers so that atomic consistency can be achieved. Each phase obeys the same algorithmic pattern, involving both abstractions, namely, a persistent broadcast followed by a quorum-based synchronization. Thus, the phases proceed as follows.

– Phase 1. First, p_i builds an identity for its requests concerning this write. This identity is the pair (i, sn_i). Then, it broadcasts a request to the servers with the goal of obtaining the timestamp associated with the last value of the object. This corresponds to line 2, where the field "no" in the message means that p_i does not need the last value of the object.

  The type of this first broadcast is defined by the pair (*cd_req*,i) where *cd_req* is the message tag, and i the sender id. Then, p_i waits until it receives acknowledgments from the processes defining a CD quorum (lines 3–6). Due to the bounded lifetime intersection property of quorums (as can be seen in the proof in the full version of this paper [13]), p_i can then define the new timestamp ts associated with the value v it wishes to write. This timestamp is greater than all the timestamps associated with values previously written.

– Phase 2. During this phase, p_i broadcast to the servers a new request carrying the pair (ts, v) (line 8). This request is tagged *write_req* and its type is the pair (*write_req*,i). Next, p_i waits until it has received acknowledgments from the processes defining a VAL quorum (lines 10–13). When this occurs, it knows (see the proof in [13]) that "enough" servers have received the write request and, consequently, the current write can terminate (line 14).

```
operation WRITE_i (x, v)
      % Phase 1 (lines 1-7): synchronization to obtain consistent information %
(1)   sn_i ← sn_i + 1; ans_i ← ∅;
(2)   prst_broadcast cd_req(i, sn_i, no);
(3)   repeat
(4)     wait for a message cd_ack(sn_i, ts) received from s;
(5)     ans_i ← ans_i ∪ {s}
(6)   until (Q_cd ⊆ ans_i);
(7)   ts.clock ← max of the ts.clock fields received +1; ts.proc ← i;
      % Phase 2 (lines 8-14): synchronization to ensure atomic consistency %
(8)   prst_broadcast write_req(i, sn_i, ts, v);
(9)   ans_i ← ∅;
(10)  repeat
(11)    wait for a message write_ack(sn_i) received from s;
(12)    ans_i ← ans_i ∪ {s}
(13)  until (Q_val ⊆ ans_i);
(14)  return()
```

Fig. 1. Implementing a WRITE () operation

```
operation READ_i (x)
      % Phase 1 (lines 1-7): synchronization to obtain consistent information %
(1)   sn_i ← sn_i + 1; ans_i ← ∅;
(2)   prst_broadcast cd_req(i, sn_i, yes);
(3)   repeat
(4)     wait for a message cd_ack(sn_i, ts, value) received from s;
(5)     ans_i ← ans_i ∪ {s}
(6)   until (Q_cd ⊆ ans_i);
(7)   ts ← max of the ts received; v ← value field associated with ts;
      % Phase 2 (lines 8-14): synchronization to ensure atomic consistency %
(8)   prst_broadcast write_req(i, sn_i, ts, v);
(9)   ans_i ← ∅;
(10)  repeat
(11)    wait for a message write_ack(sn_i) received from s;
(12)    ans_i ← ans_i ∪ {s}
(13)  until (Q_val ⊆ ans_i);
(14)  return(v)
```

Fig. 2. Implementing a READ () operation

5.3 Implementing a READ (x) Operation

The protocol for a read operation is structurally the same, and semantically nearly the same, as the write protocol. It has two phases with exactly the same meaning, as described in Figure 2. The only noteworthy difference with respect to the write protocol lies in the fact that the last field of the message *cd_req*() broadcast at line 2 carries the value "yes". This is to demand each server that sends back an acknowledgment to provide not only its last timestamp but also the associated value. This is required because a read has to return a value when it terminates (line 14).

The second phase of the read protocol is to ensure atomicity. It prevents two sequential read operations from obtaining inconsistent values. More precisely, let $R1$ and $R2$ be two read operations such that $R2$ starts after $R1$ is finished, and both $R1$ and $R2$ are concurrent with a write operation W that updates the object x from $v1$ to $v2$. The second phase prevents what is called "new/old" inversion, namely, it is not possible for $R1$ to read $v2$ while $R2$ would obtain $v1$. The prevention of new/old inversions is what makes an "atomic" object distinct from a "regular" object [23][7].

5.4 Read/Write Protocol: The Server Side

Each server s manages two local variables, ts_s and $value_s$, that contain the highest timestamp value that s has ever received, and the associated value, respectively.

As we have seen, only messages of the type (*cd_req*,i) or (*write_req*,i) can be delivered to a server s. These messages have been broadcast by p_i during the first phase (line 2) or the second phase (line 8) of the write or the read protocol.

- When a server receives *cd_req*$(i, sn, bool)$, it sends back to p_i an acknowledgment (carrying the same sequence number sn so that p_i does not confuse all acks it receives), plus the required control information (local timestamp) with the associated value if it is required.
- When a server s receives *write_req*(i, sn, ts, v), is first updates its local data if they are out of date. In all cases, s sends back an acknowledgment to the process p_i that initiated the broadcast.

It is interesting to notice that an application process communicates anonymously with the set of servers using the persistent reliable broadcast primitives. That is, an application process sees only a service and does not know the servers on an individual basis. Differently, a server works on a responsive mode, and can always send back an acknowledgment to the sender of the message it receives. The acknowledgments are one-to-one. The broadcasts are one-to-all.[8]

5.5 Another Implementation for a READ (x) Operation

There are distinct ways to implement the second phase of the read protocol. An alternative approach consists of asking the servers to inform the reader p_i when they have

[7] The interested reader can find an elaborate discussion on this difference in [16].

[8] Let us remind the reader that when we say "all the servers", we mean the set of alive servers that currently implement the desired object.

```
(1) when cd_req(i, sn, bool) is delivered:
(2)       if (bool =yes) then val_to_send ← value_s else val_to_send ← ⊥ end_if;
(3)       send cd_ack(sn, ts, val_to_send) to i

(4) when write_req(i, sn, ts, v) is delivered:
(5)       if (ts > ts_s) then ts_s ← ts; value_s ← v end_if;
(6)       send write_ack(sn) to i
```

Fig. 3. Processing by a server s of the messages it receives

stored a value whose timestamp is equal to or higher than ts (the timestamp of the value read by p_i). When p_i learns that a VAL type quorum of servers have stored such a timestamp, it can terminate the read operation and return v (the value associated with ts). Protocols based on a similar approach are described in [1, 2] to implement atomic variables from a fixed set of crash-prone disks.

Adapting this idea to our context can be done as follows. A new message tag is used by a read operation and its lines 8-13 are replaced by the following lines:

```
8'  prst_broadcast read_req(i, sn_i, ts, v);
9'  ans_i ← ∅;
10' repeat
11'     wait for a message read_ack(sn_i) received from s;
12'     ans_i ← ans_i ∪ {s}
13' until (Q_val ⊆ ans_i)
```

The code of a server is modified accordingly, namely, it additionally includes the following statement to process *read_req ()* messages:

```
(7) when read_req(i, sn, ts) is delivered:
(8)       wait until (ts_s ≥ ts);
(9)       send read_ack(sn) to i
```

Proof: Due to space limitation, the proof appears in the full version of this paper [13].

6 Practical Instantiations

Read/write objects are a general abstraction that can be used to implement various distributed services. These include, e.g., distributed shared memory, maintaining distributed files, distributed directory lookup services, shared bulletin boards, etc.

With proper assumptions about the rate of failures (process crashes), joins, and leaves, it is possible to implement the required quorum oracles with many existing *distributed hash tables*-based *peer-to-peer* systems (e.g., CAN [28], Chord [31], Pastry [30], Tapestry [34], to name a few). Specifically, most of these peer-to-peer systems provide a service that enables implicit routing of messages to servers without the application ever knowing the identifiers of the servers. The way these services operate is that the application passes an object identifier to the service. The service calculates a hashed identifier, and gradually forwards the message between some of the servers until it reaches the server whose hashed identifier value is closest, under some metric, to the hashed object identifier.

When there are no changes in the system (i.e., no failures, no joins, and no leaves), the service ensures that all requests to route a message with the same object identifier x will reach the same server. In the rest of this section, we refer to such a server as the *responsible server for object* x. Moreover, asymptotically, these systems provide with high probability good load balancing for the division of object identifiers to corresponding responsible servers. That is, when there are "enough" servers and "enough" object identifiers, each server is responsible for roughly the same number of objects. Moreover, two slightly different object identifiers (e.g., the Hamming distance between their binary representation is small) have different responsible servers.

If we assume that the rate of change in the system is low, we can employ the following scheme, similar to what is done in [4]: for a given object identifier x and constant k, we define the following *set of derived object identifiers* $\{1\_x, 2\_x, \ldots, k\_x\}$. This set of derived object identifiers implies a corresponding set of *derived responsible servers*. Thus, the set of servers that implement a shared object x now becomes the set of derived responsible servers for x.

Let us further assume that the rate of change, the latency of messages, and the speed of processes are such that there exist constants k and f so that for every set of derived servers whose size is k, at most f fail during an interval (an execution of a read or write operation). With these assumptions, $STABLE(I)$ becomes the set of derived responsible servers for object x that do not fail or leave during I. Moreover, an implementation of the oracle can periodically use the peer-to-peer service to find the current set of derived responsible servers for x, and return any subset of them of size at least $(k - f)$.

Note that the choice of k and f, as well as the assumptions about the rate of change in the systems are dependent on the specific peer-to-peer system used, as well as other external environmental assumptions. This highlights the benefits of our approach, since we have identified generic abstractions and devised a generic protocol based on them. The specification of the protocol is independent of the low level assumptions needed to implement the abstractions. Similarly, the proof of correctness only relies on the functional properties of the abstractions, and does not rely on system dependent parameters.

7 Conclusion

This paper has investigated two matching abstractions suited to the implementation of atomic objects in a dynamic distributed system where servers can dynamically enter and leave the system (or crash). One of these abstractions concerns quorum systems, the other one communication. Both abstractions are complementary in the sense they address the two basic problems encountered when implementing atomic objects (data persistence and data consistency). Their conceptual simplicity is a great advantage that allows coping with and mastering the complexity of dynamic systems. As their definition is based on abstract properties (and not on low-level assumptions), they are problem-oriented and versatile.

A read protocol and a write protocol based on these abstractions have been described and proved correct. The properties defining these abstractions can be seen as requirements that are sufficient for implementing a dynamic storage service. Instantiating the proposed abstractions in different contexts (e.g., settings defined by specific

assumptions on failures, synchrony, message delays and processing times) provides as many system specific protocols. It has also been shown that these abstractions can be realized in dynamic peer-to-peer systems satisfying appropriate requirements.

As a server has to return values and execute *Compare-&-Swap*-like operations (i.e., store a value only if its timestamp is newer than the existing one) , it can actually be either a process node or an active disk. It is consequently possible to envisage a hybrid dynamic server system made up of nodes and active disks.

References

1. Aguilera M.K., A Pleasant Stroll Through the Land of Infinitely Many Creatures. *ACM SIGACT News, Distributed Computing Column*, 35(2):36-59, 2004.
2. Aguilera M.K. and Gafni E., On Using Network Attached Disks as Shared Memory. *Proc. 21th ACM PODC*, ACM Press, pp. 315-324, 2003.
3. Alvisi L., Malkhi D., Pierce E., Reiter M and Wright R.N., Dynamic Byzantine Quorum Systems, *Proc. IEEE Conf. on Depend. Syst. and Networks (DSN'00)*, pp. 283-392, 2000.
4. Anceaume E., Friedman R., Gradinariu M. and Roy M., An Architecture for Dynamic Scalable Self-managed Transactions. *Proc. 6th International Symposium on Distributed Objects and Applications*, LNCS # 3291, pp. 1445-1462, 2004.
5. Attiya H., Bar-Noy A. and Dolev D., Sharing Memory Robustly in Message Passing Systems. *Journal of the ACM*, 42(1):121-132, 1995.
6. Chandra T.D. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.
7. Delporte-Gallet C., Fauconnier H. and Guerraoui R., Shared memory vs Message Passing. *Tech Report* IC/2003/77, EPFL, Lausanne, December 2003.
8. Delporte-Gallet C., Fauconnier H. and Guerraoui R., Hadzilacos V., Kouznetsov P. and Toueg S., The Weakest Failure Detectors to Solve Certain Fundamental Problems in Distributed Computing. *Proc. 23rd ACM PODC*, pp. 338-346, 2004.
9. Ezhilchelvan P., Helary J.-M. and Raynal M., Building TMR-Based Reliable Servers Despite Bounded Input Lifetime. *Proc. 7th European Parallel Computing Conference (Europar'01)*, Manchester (UK), LNCS # 2150, pp. 482-485, 2001.
10. Friedman R., Using Virtual Synchrony to Develop Efficient Fault Tolerant Distributed Shared Memories. Technical Report 95-1506, Dept. of Computer Science, Cornell University, 1995.
11. Friedman R., Mostefaoui A. and Raynal M., Asynchronous Bounded Lifetime Failure Detectors. *Information Processing Letters*, 94:85-91, 2005.
12. Friedman R. and Raynal M., On the Benefits of the Functional Modular Approach in Distributed Data Management Systems. *Proc. SRDS'04 IEEE satellite Workshop on Dependable Distributed Data Management (WDDDM'04)*, IEEE Computer Press, pp. 1-6, 2004.
13. Friedman R., Raynal M. and Travers C., Two Abstractions for Implementing Atomic Objects in Dynamic Systems. *Tech Report #1692*, IRISA, University of Rennes 1 (France), 2005.
14. Garcia-Molina H. and Barbara D., How to Assign Votes in a Distributed System. *Journal of the ACM*, 32(4):841-860, 1985.
15. Gifford D.K., Weighted Voting for Replicated Data. *Proc. 7th ACM Symposium on Operating Systems Principles (SOSP'79)*, ACM Press, pp. 150-162, 1979.
16. Guerraoui R. and Raynal M., Fault-Tolerance Techniques for Concurrent Objects. *Tech Report # 1667*, 22 pages, IRISA, Université de Rennes 1 (France), December 2004.
17. Hadzilacos V. and Toueg S., Reliable Broadcast and Related Problems. In *Distributed Systems*, ACM Press (S. Mullender Ed.), New-York, pp. 97-145, 1993.

18. Helary J.-M., Mostefaoui A. and Raynal M., Interval Consistency of Asynchronous Distributed Computations. *Journal of Computer and System Sciences,* 64(2):329-349, 2002.
19. Herlihy M.P., Dynamic Quorum Adjustment for Partitioned Data. *ACM Transactions on Database Systems*, 12(2):170-194, 1987.
20. Herlihy M.P., Wait-Free Synchronization. *ACM TOPLAS*, 13(1):124-149, 1991.
21. Herlihy M.P. and Wing J.L., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.
22. Lamport, L., Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558-565, 1978.
23. Lamport L., On Interprocess communication. Part I: Formalism. Part II: Algorithms. *Distributed Computing*, 1-2(2):87-103, 1986.
24. Lynch N.A. and Shvartsman A.A., RAMBO: a Reconfigurable Atomic Memory Service for Dynamic Networks. *Proc. 16th Int'l Symposium on Distributed Computing (DISC'02)*, Springer-Verlag LNCS #2508, pp. 173-190, 2002.
25. Malkhi D. and Reiter M., Byzantine Quorums Systems, *Dist. Comp.*, 11(4):203-213, 1998.
26. Martin J.-P. and Alvisi L., A Framework for Dynamic Byzantine Storage, *Proc. IEEE Conf. on Dependable Systems and Networks (DSN'04)*, pp. 325-334, 2004.
27. Merritt M. and Taubenfeld G., Computing Using Infinitely Many Processes. *Proc. 14th Int'l Symposium on Distributed Computing (DISC'00)*, LNCS #1914, pp. 164-178, 2000.
28. Ratnasamy S., Handley M., Francis P. and Karp R., A Scalable content-Addressable Network. *Proc. ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ACM Press, pp. 161-172, 2001.
29. Rodrigues R and Liskov B., Reconfigurable Byzantine Fault-tolerant Atomic Memory. Brief annoucement in *Proc. 24th ACM PODC*, ACM Press, p. 386, 2004.
30. Rowstron A. and Druschel P., Pastry: Scalable, Distributed Object Location and Routing for Large Scale peer-to-Peer Systems. *Proc.18th IFIP/ACM Int'l Conf. on Distributed Systems Platforms (Middleware 2001)*, Springer-Verlag LNCS #2218, pp. 329-350, 2001.
31. Stoica I., Morris R., Liben-Nowell D., Karger D., Kaashoek M.F., Dabek F. and Balakrishnan H., Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *ACM/IEEE Transactions on Networking*, 11(1):17-32, 2003.
32. Thomas R.H., A Majority Consensus Approach to Concurrency Control for Multiple Copy Database. *ACM Transactions on Database Systems*, 4(2):180-229, 1979.
33. Vitenberg R. and Friedman R., On the Locality of Consistency Conditions. *Proc. 17th Int'l Symposium on Distributed Computing (DISC'03)*, LNCS #2848, pp. 92-105, 2003.
34. Zhao B., Kubiatowicz J. and Joseph A., Tapestry: An Infrastructure for Fault-Tolerant Wide-area Location and Routing. *Technical Report UCB/CSD-01-1141*, U.C. Berkeley, 2001.