Active Quorum Systems

Alysson Bessani Paulo Sousa Miguel Correia
University of Lisbon, Faculty of Sciences – Portugal

Abstract

This paper outlines a flexible suite of object replication protocols that brings together Byzantine quorum systems registers and state machine replication. These protocols enable the implementation of Byzantine fault-tolerant applications that make minimal assumptions about the environment and that run in at most two more communication steps in almost all cases of non-favorable executions (in comparison with favorable executions).

1 Introduction

The research presented in this paper is motivated by two observations. First, despite the existence of much work on Byzantine fault-tolerant (BFT) read/write protocols (see [6] for a survey), most practical work on BFT replication (e.g., [1, 5, 7, 11]) is based on the notion of replicas as state machines that evolve in a coordinated way [15], which can be very restrictive for practical applications. Second, most practical services avoid the use of strong synchronization such as consensus protocols due to their complexity and underlying assumptions [4].

The BFT state machine replication (SMR) model is too restrictive for several reasons. First, although conceptually simple, this model makes it difficult to implement common mechanisms such as housekeeping and asynchronous messaging [16]. Second, most SMR protocols inherently ensure linearizability [9], which is a very strong consistency model that is not required by many applications. Third, replica determinism can be sometimes difficult to enforce, especially when one tries to implement multi-threaded services to take advantage of the multiple cores present in most modern processors. Finally, SMR requires the resolution of the well-known consensus problem, which can only be solved in a fault-tolerant way with some timing assumptions that can be subverted by malicious attacks.

Most modern (crash fault-tolerant) distributed services do not rely on pure SMR for their operations. In a recent workshop [4], speakers from eBay, Microsoft and several other companies that run large distributed systems advocated that consensus and other forms of strong synchronization can lead to several problems, and thus, their use should be avoided at all costs. The result of this avoidance is the embrace of eventual consistency models, which allow the implementation of robust and scalable distributed systems. However, the fact that weak consistency was the solution for the burden of synchronization does not contradict the fact that strong consistency is not only a nice property (it makes programming much easier), it is sometimes fundamental (some applications do require it).

These observations have lead us to the following question: *Would it be possible to build dependable and consistent services in a well-disciplined way relying on consensus only when it is absolutely necessary?* To answer this question positively, one needs options for developing dependable distributed systems apart from SMR. As far as we can see, there are two options: the first is to *increase the abstraction level* and use BFT SMR to build *coordination services* that can be used judiciously to synchronize processes in distributed systems (e.g., DepSpace [3] or Zookeeper [10]). The second option is to *lower the abstraction level* and use something more powerful than basic read/write registers, but less restrictive than SMR.

In this paper we follow this second option and propose *active quorum system objects* (AQS objects), an intermediate abstraction that can be used to implement dependable services in a flexible way. An AQS object implements a BFT register that provides standard read and write operations and, additionally, has synchronization power to implement any atomic operation (even non-deterministic ones). This unique design, gives some nice characteristics to AQS when compared with other replication protocols:

Minimal Assumptions. Since assumptions can be violated by a malicious adversary, the safe way to develop a dependable system is to assume as little as possible to make the protocols satisfy the service requirements. AQS separates operations that change the system state in write and read-modify-write (rmw), which allow systems to rely on a consensus protocol only when it is absolutely necessary (only when rmw is needed). Moreover, contrary to BFT SMR (e.g., [5, 11]), the read protocol employed by AQS never requires consensus to complete[1].

[1] A common optimization on BFT SMR makes it possible to issue a read operation to all replicas and wait for $2f+1$ matching replies, but if not enough equal replies are obtained the read must be reissued through the normal ordering/consensus protocol to ensure linearizability [5].

Stability. Another unique property of AQS is that its three protocols require at most two extra communication steps (a round-trip) when executed in non-favorable conditions (concurrency or malicious servers, excluding the case of a faulty leader replica). This unique feature makes the protocol highly resilient to malicious activity and useful for implementing intrusion-tolerant services, contrarily to other optimistic BFT protocols.

Flexibility. AQS objects can be used to implement services ranging from single-writer/multi-reader regular storage to (deterministic) state machine replication. One of the key design principles is to use the specification of the service being developed to guide the selection of the set of protocols to be used. The object operations can be implemented using a combination of read, write and rmw protocols considering access control restrictions (single-writer vs. multi-writer objects), consistency guarantees (atomic or regular register) and types of Byzantine faults being tolerated (malicious or not). If one takes into account the *specific application requirements*, it is possible to implement AQS objects that make use of the minimal protocols required to fit the application needs. Moreover, our design philosophy advocates the division of the service in as many AQS objects as possible, each one using the protocols that suit the needs of its part of the application state. This feature allows the implementation of wait-free services without using consensus in most of the supported operations.

In summary, we advocate the implementation of dependable services using multiple objects, possibly deployed in different machines, supporting operations with different semantics and requirements. The paper makes it possible by introducing the notion of AQS objects.

2 System Model

We assume a fully connected networked system with $3f+1$ servers in which at most f can fail in a Byzantine way. An arbitrary number of Byzantine-prone clients interact with these servers.

Since we rely on a modified version of PBFT [5] to execute some operations, we require partial synchrony to ensure liveness: there is an unknown instant after which all communications and computations are synchronous (with unknown time bounds).

To make our algorithms simpler, we assume that all communications are made through authenticated reliable FIFO channels. This type of channel can be implemented over unreliable fair links. Finally, we assume that all clients and servers are able to generate and verify digital signatures.

3 Active Quorum Systems

AQS is an hybrid replication model in which some operations are executed using quorum-based protocols while other use agreement-based protocols. A key feature of AQS is that object operations are divided in three classes that are implemented through different protocols:

- **write**: the state of the object is (over)written by the argument of the operation;
- **read**: the state of the object is read;
- **read-modify-write (rmw)**: the state of the object is modified according to both the operation arguments and its current state.

Write and rmw operations are collectively called *update operations*. The main difference between these two types of operations is that in the first, the state of the object is updated to the value being written (independently of the previous value) while in the second the resulting state depends on the arguments of the operation and its previous state. For example, the operation "$x \leftarrow 2$" is a write while the operation "$x \leftarrow x+2$" is a rmw (read x, update its value by adding 2 and store this new value in x). Notice that the modification done on the state is completely arbitrary and dependent of the semantics of the object being implemented.

Given these three classes of operations, AQS uses quorum-based protocols to implement read and write operations efficiently. Operations of the class rmw, on the other hand, require more expensive consensus-based protocols that are less efficient in at least two aspects: *(i.)* they require synchrony assumptions to terminate (read and write quorum protocols can be implemented in completely asynchronous systems), and *(ii.)* they usually have $O(n^2)$ message complexity instead of the usual $O(n)$ exhibited by read and write quorum protocols. This means that if the replicated object supports only operations with read and write semantics, AQS behaves like an atomic register protocol for f-dissemination Byzantine quorum systems [14], while if the object supports only general rmw semantics, the system operates like PBFT state machine replication algorithm.

Read and write quorum protocols do not fit directly with a SMR algorithm, so, we had to develop techniques to combine these two approaches in a single replication algorithm. There are two challenges that must be addressed when these two techniques are integrated to be used together. First, there has to be some mechanism that allows quorum-based read protocols to obtain a response when SMR update protocols are being executed. The main problem is that read protocols are developed to work concurrently with write protocols since both are

basic quorum-based abstractions [14], so we have to augment these protocols to be able to operate concurrently with agreement-based update algorithms. Second, we have to extend the SMR protocol used in the rmw operations to be able to execute operations even in replicas with different states, e.g., due to a write being executed concurrently with a rmw. To cope with the first challenge, we use timestamps and some properties of quorum systems to be sure that reads concurrent with rmw operations do not impair the replicated object linearizability. The second challenge is addressed through modifications on the PBFT protocol to make the primary indicate both in which state the proposed rmw operation and its proposed result together with the operation sequence number.

3.1 Protocols

In this section we outline the AQS protocols for a single object. To support multiple objects it is necessary to associate object identifiers to each variable and message handled in the protocols. The specification and correctness proof of the protocols can be found in [2].

Figure 1 illustrates the protocols execution. It is worth to mention that AQS read and write protocols are basically the BFT-BC quorum protocols [13] with minor modifications. Moreover, the rmw protocol is built over PBFT [5]. Our main algorithmic contribution is to make these protocols work together.

Write. The protocol runs in three phases, as illustrated in Figure 1(a). The first phase comprises the client reading the current timestamp from the servers and choosing the one with greatest value. In the second phase the client tries to commit a timestamp one unit greater that the greatest timestamp read on the first phase with the digest of the value to be written. This is done by sending this pair to the servers and waiting for $2f+1$ signed replies. These replies form an *update certificate*, showing that $2f+1$ servers accepted the value-timestamp pair. This certificate makes the state of the object *self-verifying*, allowing the use of only $3f+1$ servers even with faulty clients [14]. The third phase is the write itself: the client sends the value, timestamp and update certificate to the servers, which update the state of the object if the timestamp is greater than the one already stored and the certificate is valid for the value-timestamp pair.

An optimization of this protocol is to execute the second phase only if different servers reply different timestamps in the protocol's first phase. This can only happen if there are concurrent writes being executed or if some server is faulty and replies an old timestamp. In this case the update certificate will be built based on $2f+1$ replies containing the same timestamp and a digest of the value to be written (sent by the client) received on the first phase. This certificate vouches for the writing of the value with a timestamp one unit greater than the one read.

Read. The read protocol works as follows. First the client sends a message to the servers asking for the value, timestamp and update certificate of the object. It waits for $2f+1$ replies with valid certificates and chooses the value associated with the greatest timestamp. To satisfy linearizability, the client must ensure that all subsequent reads that happen before some update is executed will read the same value observed by it. This is done by writing back the read value in the servers that are storing values with lower timestamps.

Figure 1(b) presents the message pattern of AQS' read protocol. Notice that no writebacks are needed in fault- and concurrency-free executions.

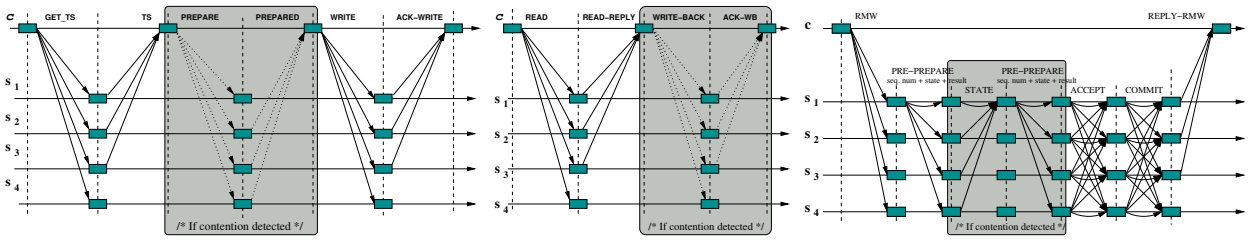
Read-Modify-Write. The rmw protocol extends PBFT [5] taking advantage of two of its features: *(i.)* the existence of a primary and the capability to elect a new one in case it is faulty and *(ii.)* the ordering of operations, required for correct execution of rmw operations.

In order to execute a rmw operation *op* on the system, a client sends a signed request containing *op* to all servers and waits for $2f+1$ reply messages from different servers with the same result *r* for *op*.

Upon receiving the rmw request from the client, and before ordering and executing it, the primary should *read* the state of the object: each server sends the object state (value, timestamp and last update certificate) to the PBFT primary. The primary waits for $2f+1$ of these messages and chooses the value associated with the greatest timestamp received as the operation base state. The $2f+1$ received messages are used to build a *base state certificate* that justifies the choice of the state in which *op* will be done. After that, the primary starts the execution of PBFT with the following modifications:

1. Before ordering the rmw request, the primary executes *op* on its base state and generates the update state and the result of the operation. The "rmw request" to be ordered by the primary (using PBFT) is composed by the base state, the base state certificate, the requested operation, the update state (or the difference between this state and the base state) and the operation result. A sequence number should be assigned by the primary to this message and disseminated to the other servers on a PBFT PRE-PREPARE message.

2. Each server only accepts a PRE-PREPARE message for an rmw if, besides the usual conditions defined in PBFT [5], the base state is justified by the base state certificate, and the update state and operation reply



(a) Write. (b) Read. (c) RMW.

Figure 1: AQS protocols. Grey phases are only executed in cases of update contention or in the presence of malicious servers.

are a *possible* resulting state and operation result of the execution of *op* on the base state. If the PRE-PREPARE message is not accepted, the primary is then suspected by the server.

3. The PBFT COMMIT messages must contain a digest of the update state and must be signed to be used to build update certificates.

4. After committing the rmw request, each server will define the update timestamp as one unit greater than the timestamp associated with the base state (it can be obtained from the base state certificate). After that, it will verify if this timestamp is greater than the current stored timestamp and, if indeed the timestamp is greater, it updates the object with the update state, timestamp and the certificate (composed of $2f + 1$ COMMIT messages). In any case, the server sends a reply to the client with the result of the operation *op*.

5. When a new primary is elected (to substitute some previous primary that was unable to order some operation *op*), each server sends its object state together with its PBFT VIEW-CHANGE message. The new primary then can process the operation *op* to update its current state choosing the valid state with greater timestamp t_s among the received states.

One possible optimization for this protocol is to exploit the case in which there is no write concurrent with the rmw operation, if expected to be common. In this case, the first phase of the protocol, in which the primary collects object states from other servers does not need to happen. Instead, when the primary receives the rmw operation from the client, it can use its object state as the operation base state and the base state certificate as the update certificate for this state. Other servers will accept this base state if the base state certificate is valid and the timestamp is greater or equal to their current timestamp. If the message is not accepted, the system will revert to the common execution. Figure 1(c) illustrates a fault-free execution of the rmw protocol with this optimization (see details in [2]).

3.2 Extensions

The basic protocols can be extended to avoid the cost of public key signatures and to support multi-object operations, as discussed bellow.

Avoiding Signatures. To avoid the costs of public-key signatures, the AQS protocols can also be changed to use only MAC vectors (a.k.a., authenticators), in a very similar way to what is done in HQ [7]. However, the consequence is that the protocol will need to address several corner cases, which will make it more complicated. In order to maintain the stability of having only two more communication steps in non-favorable conditions, all protocols' signatures can be changed to MAC vectors, with the exception of the first phase of the write and the read phase of the rmw. This will impose the costs of signatures for all update operations, but there are indeed some advantages in doing that: (1.) the protocol is exactly the same and no corner cases to address invalid authenticators have to be implemented; (2.) modern machines have many cores that can be used to both verify and generate signatures in parallel; (3.) if the objective of the system is to tolerate non-malicious Byzantine faults (e.g., heisenbugs, memory, network and disk corruption), the signature can be implemented with a simple digest of the message plus the same unique id associated with the signer, which imposes no performance drop on the protocol.

Multi-object operations. Since AQS strongly advocates the partition of the service state in as many objects as possible, it is natural to have some multi-object operations on the system. AQS deals with them using two simple rules: (1.) if some single-object operation is a rmw, then the whole multi-object operation is executed as a rmw; (2.) if some of the single-object operations are reads on objects that will be used to write on (possibly different) objects, the multi-object operation must be executed as a rmw. These rules solve the problem if all involved AQS objects are deployed on the same set of servers and use the same primary for processing their rmws. The processing of multi-object operations when some of these constraints are not satisfied is left as future work.

4 Weakening the Protocols

The kind of reasoning used to develop AQS can be exploited to make BFT protocols that are simpler (when compared with other protocols, e.g., [1, 7, 11]) and more robust (in the sense that the protocols requires less assumptions to terminate). Many recent BFT protocols are highly optimized for the expected common-case in which there is synchrony, no faults, and no contention on object access. All of these optimistic conditions are part of the environment and can be influenced by an adversary that wants to attack the system. So, the question that one can raise is: Is it possible to build "optimized" protocols that do not assume conditions on the environment? The answer is positive, so instead of expecting that *assumptions* about the environment hold, the protocol can explore *knowledge* about the applications that will use it.

The environment comprises the network, the machines and processes that interact with the replicated system. Some common assumptions that are made about the environment in order to optimize BFT protocols are: no contention [1, 7], synchrony [11], no faults [1, 7, 11], etc. As already discussed, there is a danger with these assumptions because they can be attacked to degrade performance significantly.

On the other hand, the applications that run above the protocol provide great opportunities for optimizations that are yet to be explored. The fact that we have AQS instead of SMR allows exploring the optimization space considering several dimensions of the application semantics. First, the service *operation semantics* can be divided in three classes (read, write and read-modify-write) and minimal protocols can be found for any one of these classes. Second, the application data *consistency requirements* can be exploited to avoid complex protocols that ensure linearizability [9]. For example, in many applications regular semantics[2] are sufficient for read operations. Finally, the replication middleware can take advantage of the access control information to determine that some objects can only have their states modified by a *single writer*. Notice that this is very different from expecting no-contention: what we do is to use the knowledge that some objects will never experience update contention.

The protocols presented in the previous section only address the first dimension. However, these protocols can be modified and optimized to ensure less strict requirements:

- **Single-Writer Write**: if there is a single writer for a given object, there are no concurrent writes or rmw operations with this client's writes. Consequently, the two first phases of the write protocol are not necessary and the reply of the third phase of a previous write can be used as the update certificate. Therefore, the client does not need to query the system and can use the ACKs of the previous write as an update certificate for the next write.

- **Regular Read**: if in case of write-read contention the value being read can be one of the values being written or the previous value of the object, we can simplify the read protocol simply cutting the write-back phase.

- **Single-Writer RMW**: if there is a single writer, there can not be contention between update operations. It means that the single-writer rmw can be done as a simple single-writer write.

Table 1 presents the number of communication steps required for each variant of the AQS protocols. Since server faults do not affect both quorum systems (besides making the contention-free case not hold) and PBFT (if the faulty server is not the primary[3]) we did not consider these failures when we computed the communication steps.

	SR	SA	MR	MA
read	2	2(4)	2	2(4)
write	2	2	4(6)	4(6)
rmw	2	2	5(7)*	5(7)*

Table 1: Number of communication steps of the AQS protocols in cases with and without contention (when some optimization is available, the worst case value is put inside brackets). SR (Single-Regular), SA (Single-Atomic [12]), MR (Multi-Regular), MA (Multi-Atomic) are the variants of the protocol. Protocols marked with '*' require partial synchrony for liveness.

5 AQS Applications

This section provides a few examples of classes of applications with different requirements that can benefit from AQS' flexibility.

Storage. A storage service can be implemented using rmw operations on a *block manager object* to create blocks (assigning unique block ids) and reads and writes on the *block object* to access and modify its data. Further, the access permissions and the weak semantics of storage systems can be exploited to use the single-writer and regular versions of the protocols when possible.

[2] In a *regular register* [12] the result of a read that is concurrent with one or more writes can be the value of the register before the writes or one of the values being written.

[3] In case of faulty primary, PBFT performance suffers strong degradation until a correct server is elected as a leader [8].

LDAP. The *Lightweight Directory Access Protocol* specifies means to access an hierarchical name space in which every node is an entry associated with zero or more attributes. In our AQS implementation of LDAP every *entry*, *attribute* and *user* is an AQS object. The main operations supported by LDAP affect one or more of these objects using different protocols: *bind* and *unbind* requires single-writer write on the user object; *search* requires reads on many attribute abjects; *modify* requires write on one or more attribute objects; *add*, *remove* and *modifyDN* require rmw operations on entries. It is worth to notice that search and modify, by far the most used LDAP operations, do not require rmw (and thus, do not require consensus execution).

Deterministic state machine replication. The rmw protocol can be used directly to implement deterministic services that require SMR. Moreover, since we are considering deterministic services and that no writes happen, the primary does not need to execute the operation and send the update state to other replicas. In consequence, the rmw protocol will work almost exactly as the PBFT protocol. The read protocol can be used as it is to get the system state (or parts of it), without ever running an agreement protocol.

6 Conclusion

This paper outlined the basic ideas of the AQS protocol suite. The protocols here presented integrate in a natural way Byzantine quorum systems read/write protocols with BFT SMR and enable a set of unique properties such as the use of minimal assumptions (still satisfying the service specification) and at most two extra communication steps in almost all cases of non-favorable executions.

Acknowledgments. We warmly thank André Sousa and Nuno Ferreira Neves for discussions on the paper that greatly assisted us in improving it. This work was partially supported by the FCT through project PTDC/EIA-EIA/100581/2008 (REGENESYS), project PTDC/EIA-EIA/109044/2008 (ReD) and the Multiannual and CMU-Portugal Programmes.

References

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *SOSP'05*, 2005.

[2] A. N. Bessani. Active quorum systems: Specification and correctness proof. Technical Report DI–FCUL–TR–2010–2, Department of Informatics, University of Lisbon, July 2010.

[3] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga. DepSpace: a Byzantine fault-tolerant coordination service. In *EuroSys'08*, Apr. 2008.

[4] K. Birman, G. Chockler, and R. van Renesse. Towards a cloud computing research agenda. *ACM SIGACT News*, 40(2), 2009.

[5] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4):398–461, Nov. 2002.

[6] G. Chockler, R. Guerraoui, I. Keidar, and M. Vukolić. Reliable distributed storage. *IEEE Computer*, 42(4):60–67, 2009.

[7] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ-Replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI'06*, Nov. 2006.

[8] W. S. Dantas, A. N. Bessani, J. S. Fraga, and M. Correia. Evaluating Byzantine quorum systems. In *SRDS'07*, Oct. 2007.

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

[10] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for Internet-scale services. In *Usenix ATC'10*, 2010.

[11] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *SOSP'07*, Oct. 2007.

[12] L. Lamport. On interprocess communication (part II). *Distributed Computing*, 1(1):203–213, Jan. 1986.

[13] B. Liskov and R. Rodrigues. Tolerating Byzantine faulty clients in a quorum system. In *ICDCS'06*, 2006.

[14] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, Oct. 1998.

[15] F. B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[16] W. Vogels. Life is not a state-machine. Invited talk on ACM PODC'06, 2006.