Quorums Systems as a Method to Enhance Collaboration for Achieving Fault Tolerance in Distributed Systems

Ioan PETRI
Babeş-Bolyai University, Business Information Systems Dept., Cluj-Napoca, Romania
ioan.petri@econ.ubbcluj.ro

A system that implements the byzantine agreement algorithm is supposed to be very reliable and robust because of its fault tolerating feature. For very realistic environments, byzantine agreement protocols becomes inadequate, because they are based on the assumption that failures are controlled and they have unlimited severity. The byzantine agreement model works with a number of bounded failures that have to be tolerated. It is never concerned to identify these failures or to exclude them from the system. In this paper, we tackle quorum systems, which is a particular sort of distributed systems where some storage or computations are replicated on various machines in the idea that some of them work correctly to produce a reliable output at some given moment of time. Thus, by majority voting collaboration with quorums, one can achieve fault tolerance in distributed systems. Further, we argue that an algorithm to identify faulty-behaving machines is useful to identify purposeful malicious behaviors.

Keywords: *fault tolerance, quorum systems, distributed computing, byzantine faults.*

1 Introduction

Many distributed systems are designed to support resource sharing. On its traditional architecture, Internet is built to be a network of computers as it is composed by many networks. Networks of computers are everywhere and they are communicating and coordinating by message passing. Mobile phone networks, corporate networks, factory networks, campus networks, home networks, in-car networks either separately or in combination share essential characteristics. A system with networked components which collaborate in their actions needs to have the ability to work well even when the number of users is changing and they are not reliable. Reliable users' number can change due to their collaborative behavior, some of them behaving as faulty ones or because new users join the network.

Distributed systems are built to tolerate failures. In the same time they should be designed to be robust and aligned with the newest discovered standards. Finding the faulty entities of such a system might be compared with the tenant discovering problem: *any authority shouldn't be contacted before the landlord isn't sure the tenant really exaggerates in consuming resources and not paying the costs.* Modern distributed systems are designed on the fault tolerance principle. They have self-checking mechanisms to verify system status; as well they are able to take suitable actions to prevent failures. Replication is one such mechanism. With replication, either one piece of data is distributed to several storage locations, or some computation is distributed to several servers.

Many distributed systems are concurrently. Different actions happen in the same time. For data replication many processes will take place in the same time. This concurrency is the cause of many bugs. Failures fall in the following categories, from the sources that originated them [2]:

- Hardware failures, like battery loss, disks loss and hardware generations changeability. They can be solved physically and do not request the system to be redesign.
- Software failures represent the most important problem to solve inside systems. Many studies proved that even after lots of testing procedures, software is not out of unplanned problems. New programming languages offer debugging and runtime tools as very important steps for tracking down the errors. Also replacing the old manner of object reference with the shadowing technique of building object improves the software performance and removes failure risks.
- Other sources of failure account to more than one-third of system's failures. They involve planning, maintenance, backups and environmental factors like power outages, air conditioning or heating failures

To avoid such failures, virtual synchrony has been introduced. Virtual synchrony simulates the system as it wouldn't use replicated data or concurrency even if it doing so. It substitutes groups of entities with single entities and improves all designed performances.

On the same reasoning, membership is considered a distributing important feature. It identifies a set of members (system entities) with common features to build a community that is agreeing on something [6]. Membership is an important feature when working with replication, because it indicates whether a machine is part or not of a selected quorum.

2 Approaches for fault tolerance in distributed systems

Computer systems fail in different situations. The reasons of failure are changing from hardware to software; many times programs are not finishing their intended computation. On distributes environments, a failure is partial because while some components are failing while others continue to function. To handling all this failures different techniques are introduced:

- Detecting failures; those failures which can be detected and for those which cannot be detected, the solution is to manage intended processes in the presence of them
- Masking failures; those failures that can be detected and can be hidden or the severity can be reduced.
    - Message that can be retransmitted when they fail to arrive
    - Files that can be written to a pair of disks to replace the one which is corrupted
- Tolerating failures; most of Internet services are having failures but they can be tolerated instead of being detected and hidden.
- Recovery from failures; the mechanism of rollback is implemented to cover any server crashes. When a failure will occur a data inconsistent state will appear.

Software architecture has different structured levels. Modules are used as a single computer. Software architecture is expected to intermediate services offered and requested between processes located in the same or different computers. A server is a process that accepts requests from other processes. A service can be provided by one or more servers which interfere with client processes in order to maintain the consistency of service.

The *client-server architecture* represents a structure where client processes interact with individual server processes in order to access the shared resources. Servers can in turn be clients of others servers: a web engine is behaving as a client and as a server in the same time: it responds to queries from browser clients and it runs web task as a client.

Peer to peer architecture uses all processes inside an interacting cooperation. Every process is represented as a *peer* without any difference between client and server processes or the computers that they run on. Applications are built with a large number of peer processes running on separate computers. They use communication patterns. Each object is replicated in several computers to further distribute the load and to provide protection against disconnection. The peer to peer architecture is placing individual objects and maintains replicas among many computers [2].

Distributed systems always involve an operational list of possible computers. Such list requires a scan of all available peers and also very complex operations of monitoring the processes that run on the system. The purpose of scanning is to identify a replica. A replica inside distributing systems can store any important parameter or set of parameters (variable values, timestamps)**.** A distributing system static model contains an unchangeable list of members. At a particular timestamp only a subset of these members can be used – which is a quorum. Inside a static membership model, every process *p* has details about all other running system processes. The information that usually is missing for each process is the list of operational processes that run inside the system. The requests that a process *p* might send to all other processes are referring to various dysfunctions. The main objective for a process request is to get back a majority of the process responses. Reaching the majority involves member contract on a particular request. A process majority will gather all the positive feedbacks received from system entities. Usually a request is deployed during an operation of reading or writing. Both of them need to reach the majority of processes instead of satisfying some performance standards.

Read and write operations imply a set of particular processes. In order to accomplish the read and write operations, it is required to execute all the processes with a minimum number of replicas involvements. The usability is locked by the condition that some minimum number of replicas need to be read (QR) and some minimum number of copies to be updated (QW) and both of them should be bigger than the system size(n):

$$QR + QW > n$$

e.g. $QW = n - 1$, $QR = 2$ => $QW + QR > n$

QW, QR represent read and update (write) processes that run inside a quorum architecture. The particular formula allows the minimum condition for the process in updating variables even if one of the groups is faulty. Any read will discover the most current update. So a process *p* executes an update for the variable *x* how it follows:

1. Attempts a read operation to check and find the most current updated values of the variable with the associated time. A RPC read request for all replicas will take place. Remote Procedure Call (RPC) is a data flow between caller and provider that involves data movement between server and networks. The chosen version of *x* will be the one with the newest timestamp and with the associated value.
2. Process *p* establishes a timestamp when the newer version for variable *x* will be implemented. The new value for time variable will be chosen as long as it is the larger than anyone read from the group.
3. Sends another RPC to at least QW members making a news in preparing them to update x.
4. It receives a response from the group with acknowledges for the new value. Process *p* looks to see if it has a write quorum. If the number of acknowledgement is QW or larger, *p* allows the update to commit, otherwise it will be aborted, meaning that the members are not changing the replica value.

Condition: A system with QW<*n* is considered fault tolerant. This suppose that if *p* wants to read *x*, then it has to send RPC to some other processes because QR>1.

Remark: Local copies for variable *x* are built by copying a value from the volatile region data storage to a temporary area. The volatile storage area is not a very safety data storage region. If a failure happens, then all the information will be lost. The information will be stored on temporary data storage and then transferred to a persistent storage location.

3 Evaluation of the quorums approach

A very compatible protocol is continuously developed to cover any community unpredictable malicious processes. The general agreement problem involves bounded failures processed inside a tolerated environment. *Bounded* defines particular process actions that derives from any failure and it can produce a randomly unexpected behavior. Byzantine agreement algorithm is used as a future security protocol template. This model assumption is based on a limited number of failures and the severity of the failure is unlimited. Our goal is to identify the faulty participants.

The general agreement basic protocol finds a number of generals that receive an order to attack from a superior. The mandatory internal requirement is synchronization. For the decision to be taken, they need to exchange messages in order to find other generals statuses from the process. Encountered statuses are to attack or not to attack. There are *i rounds* of communication and all participants are synchronizing themselves to attack at the round *i+1*.

The main decision of attacking is adopted when all the loyal generals will have the *attack* status. The siege will when all loyal generals will have a *non attack* status. A *traitor* status is any free attitude to lie about the own state or to send any additional message. Faulty generals can never forget the message to a loyal general. After the first round of changing messages, comes a second round of voting. All participants have their own opinion about the other statuses.

Clock synchronization is a request of the process because the algorithm has to limit its execution to a timestamp period.

Lamport at al. proved that at most *t* traitors inside the community at least 3*t+1* participant can be accepted. This implies that no less then 2*t+1* loyal general should be present in order to exclude the messages delivered by *traitors.* For *t faulty participants* a set on *t+1* round it is required for changing messages.

The protocol is assuming that all participants know about other running processes. It has a very low performance in finding actually the number of faulty participant inside the process. There is an idea about this failure but actually no process is interested to identify those faults.

Replication interference is used to increase the complexity of byzantine agreement algorithm and to have a closer applicability on what nowadays technology requests.

The byzantine agreement quorum replicates an object for k^2 nodes. All the nodes are assigned representing inside a $(k \times k)$ *system.* A read quorum is implemented through a row as well as a column is a writing quorum.

The figure 1 shows a quorum with n=120 elements, d=12 columns, h=5 bands, r=2 rows per band. Based on before analyses, we can consider the column as being writing quorums and the rows to be reading quorums.

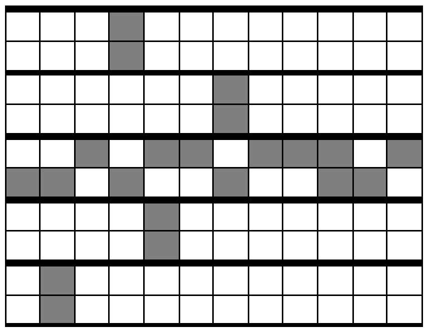
Fig. 1. Grid Quorum system

The limited byzantine agreement model is applied on a structure of faulty servers using data replication. Replicating data through a set of non-trusted servers is solved by using byzantine agreement algorithm to update read and written values [7].

3.1 The quorum approach

For our approach, a community is considered a quorum. Quorums have appeared from the necessity of a closer applicable security around byzantine agreement model. A default quorum is designed for a population of $P = \{P_2 \dots P_n\}$ a set of servers with $\delta \in 2^p$ a set of subsets of P.

Condition1: Each Q belonging to δ is a quorum ($Q \in \delta$).

Condition2: Every subset P_i is an interactional entity.

Process: A variable is replicated inside the system. A client can be a reader or a writer and it has assigned a particular *t* timestamp. Every involved server P_i stores local copies of x_i and t_i.

• WRITE (to write x); A quorum is chosen by a writer. A writer can be any server or set of servers from *P*. the write quorum evolves as follows [1]:

- o The writer increments the timestamp indicator t and sends (write, x, t) sets to all P_i from Q;
- o A set of (write, x, t) having $t > t_i$ is received where t is the newest timestamp assigned for writing by the most recent process and t_i is the last timestamp of the process before receiving this request of writing,
- o server P_i sets (x_i, t_i) <- (x, t) and returns an acknowledge message.

• READ (to read x): A quorum is chose by a reader. A message of read is sent to all P_i from Q. The read quorum evolves as follows:

- o P_i returns (value, x_i , t_i)
- o the reader waits the values from all servers from Q
- o Selects the one with the highest timestamp.

3.2 The byzantine quorums

A byzantine quorum is a special case of a quorum system that tolerates byzantine failures in a random population of servers' subsets with a property of interconnectivity. The Premise is that are enough correctly servers to guarantee consistency of the replicated data.

On byzantine quorum system, the process uses a central server element (the same time we can consider it as client) which is evaluated on its default behavior. All servers respecting particular specifications are considered to be correct. Any other deviated behavior is considered *faulty.*

The assumption is a population β which contains all fail prone systems and a subcategory B belonging to β. The only way a client can get the correctness of the accessed server from a quorum is to access every server. Multiple quorums operations are allowed [3]. The basic operations of reading and writing on quorums are the same as for the traditional quorums. Systems are often failing. Most of failures are partial and can be detected. To hide failures byzantine quorums deploys the following types [3]:

1. Masking quorum systems: which are dissemination quorum systems and opaque masking quorum systems? On masking quorums system load becomes a very important issue. Quorum system load over *n* servers have a load of $O\left(\frac{1}{\sqrt{n}}\right)$. A masking quorum system for a fail prone system δ delivers two main properties:

- M-consistency:

$$\forall Q_1, Q_2 \in \delta \; \forall B_1, B_2 \in B: \quad (Q_1 \cap Q_2) \setminus B_1 \not\subset B_2$$

- M – Availability:

$$\forall B \in B \exists Q \in \delta: \quad B \cap Q = \phi$$

A dissemination quorum system for a fail prone system δ delivers two properties:

- D-Consistency:

$$\forall Q_1, Q_2 \in \delta \; \forall B \in \delta: \quad Q_1 \cap Q_2 \not\subset B$$

- D-Availability:

$$\forall B \in \delta \; \exists Q \in \delta: \quad B \cap Q = \phi$$

The central idea of masking quorum system is to mask any faulty behavior of data repositories. Figure 2 depicts a masking quorum system. On the figure 2,

B - Represents random set of faulty servers

Q1 - is a quorum used to write one arbitrarily *x*

variable
Q2 – is a quorum used to read one arbitrarily *x* variable

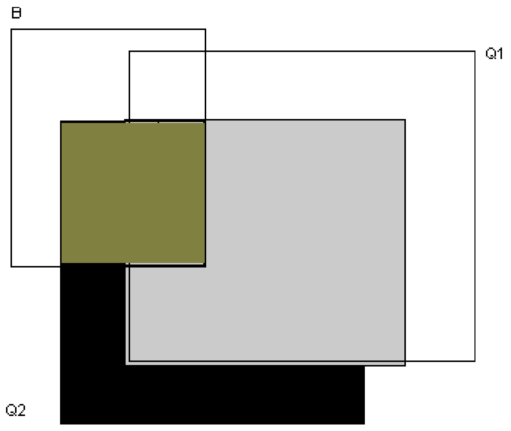

Fig. 2. Masking Quorums system

A dissemination quorum system involves clients which can operate modifications on the faulty servers. On quorum's auto-behavior there is an internal process of self-verifying written replicas. They will be transmitted to all reading operation from the system, despite any arbitrarily failure of some servers.
Malicious intentions can affect the algorithm status. Clients can leave an operation to an inconsistent level, where only few of the steps are completed. *For instance a client due to its intention of writing on x variable only sends notification to processes to announce its following operation without completing the process. If any failure of the system is recorded the cause won't be the involved server any more.*

3.3 The byzantine fault tolerance

Byzantine fault tolerance represents any robust behavior that prevents system's failure. Traditional byzantine tolerant behavior accommodated a maximum number of *t* traitors; *f*or any *t* numbers of traitors at least *2t+1* needed to be honest peers and *t+1* steps of message changing. Fault tolerance defines the same robustness showed on the traditional system capacity of supporting reliable reading and writing operations.
A fault is defined to be a physical defect that can happen in different parts of a system. A fault is considered to be mostly an error because an error is the beginning of every failure. Every failure has a number of attributes like cause, duration, nature, extend, value and also it has a type. A permanent fault continues to exist until it will be fixed and a transient one occurs and then disappears. A fault that disappears with an identified frequency is called intermittent failure.
Because of their multiple inconveniences many techniques to deal with fault have been deployed.

- Fault avoidance prevents the occurrence of faults, e.g. Quality control: design review, component screening.
- Fault masking prevents faults from introducing errors e.g. error correcting codes, majority voting.
- Fault tolerant system is a system that continues to function correctly in the presence of failures. Such a system includes: fault detection, location, containment and recovery.

Inside a particular system failures take place with a specific rate. For any ℓ rate a mean time to failure (MTTF) parameter is established. This parameter defines the expected time when system will operate before the first failure occurs; MTTF=1/ ℓ;
On the failure context there are many techniques for redundancy in order to allow fault detection, fault masking or fault tolerance: [11]

- Forward recovery helps the system to continue the operation with the current system state even if it may be faulty.
- Backward recovery uses previously saved corrected state information at the starting point after failure. Avery time the process begins a copy of the initial data is stored. A record with all transactions are made a subsets are saved at specific points. This recovery technique uses *check pointing* when the system roll back in order to recover from failure after repair. The state of the first process is recorded on a double storage module. There is also an analog procedure of *checkpoint message* when the secondary process gets its current state from the most recent message. The *persistent* technique implements the states as transactions and undoes any recent uncommitted state.
- Use of recovery blocks
  - o Execute critical functions
  - o Test the output after each execution
  - o Invoke the acceptance test upon detection of failure
  - o Recovery blocks that works with a watchdog timer to initiate processes based on previous acceptable results

In the byzantine fault tolerance, reading and writing replicated data becomes a very challenging operation. Any random *x* variable should be read / written anytime, by different processes against any deviated behavior. An unpredictable server failure or a client failure might need to be covered. The process is as follows:

• With the newest timestamp, a process *p* can sent notification about its intention of updating value of *x* variable. After this notification it expects feedback from all the interested processes.

• Process *p* receives a respond from the group with an acknowledgement for the new value.

• Process *p* looks to see if it has a write quorum. If the number of acknowledgements is QW (updating quorum) or larger, *p* allows the update to commit, otherwise it will be aborted, meaning that the members are not changing the replica value.

The condition for the byzantine fault tolerance to happen is $QW<n, QR >1$

Any fault tolerant system will have QW (updating *x* that usually is using a minimum number of copies) smaller than *n.* If *p* wants to read *x* it has to send RPC to some other process because QR>1.

Fault tolerance purpose is to hide as many as possible faulty servers. The following steps happen:

1. Read/Write objects are replicated to each *n* server from the system. Sometimes read methods are called *queries.* Usually we can deal with operations and queries.
2. A client performs operations to an object by issuing requests to a quorum of servers.
3. A server receives the request and then invokes a method on its object local replica. Each time the server invokes an object a new object version will result. The object version together with its timestamp is recorded by server every time.
4. All these records are store to an array and the *replica history* is built.

Client process:

Client role is to interact with servers. Interaction process can have from one to many steps of performance. The routine scenario releases one step client-server communication. To a higher level of complexity a process is never completed in one level because the client is facing during the communication process with failures.

Clients are sending operations and then receive answers from a service. On the server are running all the processes that are requested by any client. A server access is based on object history set. This object history set contains lots of data that actually is used to classify object versions that a client has to perform on servers. The quorum of servers is the only entity that posses the latest object version.

Server process:

Server mostly validates the object history set by comparing the timestamps of an object. If the timestamp for validation is not the latest one than the object validation won't pass. For opposite situation server will update himself the object timestamp and will start building an answer for the method. During the process lots of inconveniences are revealing. Any server failure brings an inherent inaccessible quorum. This situation gets the client in the position of checking for additional servers in order to collect a quorum of responses (finding a live quorum). As every operation is bounded by a timestamp only an honest client behavior will complete an operation because the servers are expecting the operation to have the same timestamp. This algorithm gives a truthful state for system because any malicious client that wants to force on server object with the same timestamp is excluded [4].

3.4 The quorum proactive recovery

Byzantine faulty replicas can be replaced and tolerated using a proactive recovery [10] mechanism. Based on the assumption that it is very low possibility for individual replicas to fail simultaneously, a quorum system keeps all its performances if it uses recovering replicas proactively.

The system will suffered no performance consequences as the recovering proactively mechanism uses a detection algorithm which identifies those replicas with a long time recovery. To solve all the implications of recovery from Byzantine faults it is needed:

• Proactive recovery is based on the assumption that a replica even it is faulty can behave properly and many times it cannot be identified. Recovering replicas proactively and periodically excludes every risk of unidentified replicas. For its default applicability proactive recovery assumes that all the recovered replicas are non faulty as well the recovery mechanism will never produce faults instead of safety.

During its recovery, replica should not be excluded from the process it was involved.

• Fresh messages protect a replica to be controlled by an attacker who knows the key for authentication. Every replica should be able to check if a message is fresh. It should exclude and reject all old messages as well as it should be able to prove that a message is authentic or not.

- Efficient state transfer improves the system's performance but it is harder to be applied with Byzantine faults. System's local copies are checked in order to determine all up to date portions of replicas. Any missing replica should be recovered using correct replicas.

The proactive recovery algorithm uses state machine replication. Several replicas that host a service are replicated on different nodes. Traditional operations allowed on the replicated service can be replaced with particular computations. Replicas are identified and they maintain a copy of server state. Different operations are performed using the replicas internal role. The property of faulty comes from algorithm's inconsistency to support impersonate replicas. Any other replicas that follows the algorithm are non faulty.

Requirements:

- replicas need to start in the same state
- replicas need to be deterministic(the operation results must be the same every time)

Algorithm: All non faulty replicas execute operation on the client requests. All the operations have same type and same order. After the request the client waits for f+1 replies from a population of 3t+1. *The algorithm goal is to guarantee that all non faulty replicas agree on a total order for the execution of requests despite failures* [10]. For the algorithm accomplishment the backup mechanism is used. Replicas are moving from a node to another, passing through different system configurations called *views.* A view is holding one primary replica and the others are backups. This procedure increases the system's performance and security. Using *views* a client will always be able to check if a replica is faulty as well as it will always find the correct replica value. The client will request operations. Every operation will be assigned with a number. When a fault is met the *view* changes and it will choose another primary replica.

3.5 The quorum replication

The byzantine algorithm tolerates faults. It makes no assumption about the faulty components, so it is supposed to tolerate many malicious attacks. The recent work proves that an invisible synchrony between entities is desired to solve byzantine algorithm limits. To cover big amount of failures the faulty components needs to be detected. Based on this, it had been discovered that for any asynchronous system this failure detector can't be very accurate, sometimes misclassifying a replica as a faulty [4].

The traditional algorithm is based on the assumption that no more than 1/3 of the group can be faulty for the algorithm correct execution. If any of non faulty replicas are considered faults, the attackers will reach the whole control of the replica without changing any detectable attribute. All faults will be excluded from the group and the number of non faulty replicas will get lower. The algorithm efficiency will decrease.

The Fleet algorithm [4] affords n>3 replicas to tolerate *f* faults, with the rule that malicious clients can have the knowledge of correct replicas. For this implementation, Fleet requires *n>4f* replicas. The same algorithm introduces a mechanism of finding the number of faulty replicas. A very pessimistic scenario assumes that clever attackers can transform dead replicas to behave correctly until the attacker controls more than *f.* In this case no algorithm can do anything.

For higher level Byzantine fault tolerance has been completed with abstract specification encapsulation. All the replicas are running on the same service implementation and they are updated in a deterministic way.

A fixed array of pairs is established containing an object and a generation number. Each object is identified with a particular identifier. The generation number is incremented every time the entry is assigned to a new object. The algorithms works with 4 particular types of objects: files, directories, symbolic links and null objects. Every null object indicates which object is free. Any other non null object will have meta-data that includes attributes.

Algorithm: A procedure is called by a client to invoke an operation on the replicated service. The client side is carried by the procedure and it will return the result when enough replicas have responded. When an operation is implemented an up call for execution of procedure is used.

A central BaseClient and BaseReplica will be built to improve system's tolerance. Any others entities involved will have to consult this base entity. Every time a replica is faulty the system will roll back and check the previous replica from BaseReplica. [10]

4 Conclusions and Future Work

Fault tolerant algorithms have been theoretically analyzed and practical improved but they never give a higher efficiency than *t* traitors from a community of *3t+1.*

This paper position is to find a trustful algorithm that survives to a higher number of failures. On the server's side many failures can happen. These failures belong to an internal process managed by an authority responsible with the server's consistency and reliability. Variable statuses can replicate themselves to different machines helping faults to spread over communities. A quorum is a community where every machine status has to be identified instead of a safety allocating resource environment.

For a better understanding, the problem can be dropped to a complex scenario where different statuses with a different replica failing level can be analyzed.

Servers can hold faulty replicas. They refuse to response with the right replica statuses starting a malicious process of colluding. Colluding represents any unpredictable behavior of responding with inappropriate set of replicas (values and timestamps (x_i, t_i)). Every quorum entity will vote at a certain process level and every fault response will examine algorithm's performances. Most algorithms are fault tolerant with an established number of deprecated behaviors. They provide rated outputs while identifying the fault replicas from the community.

Quorum analogies were made from the prospective of finding a suitable research area to implement a security based algorithm. The new algorithm we intend to develop tries to improve the previous algorithms' performances with an operational probability.

Acknowledgement

This work is supported by the Romanian Authority for Scientific Research under contract IDEI_573

References

[1] K. P. Birman, *Reliable Distributed Systems, Technologies, Web Services and Applications*, Springer Science & Business Media , LLC, 2005.

[2] G. Coulouris, J. Dollimore, T. Kindberg, *Distributed Systems*, *Concepts and design*, Pearson Education, 2005.

[3] D. Malkhi and M. Reiter, "Byzantine Quorum Systems," *AT&T Labs – Research*, Florham Park, USA.

[4] C. Cachin, "Security and Fault-tolerance in Distributed System," *IBM Zurich Research Lab*, ETHZ, 2007.

[5] M. AbdElMalek, G. R. Ganger, G. R. Goodsony and M. K. Reiter, "FaultScalable Byzantine FaultTolerant Services, Jay J. Wylie 14," In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pp. 59–74, Brighton, UK, 2005.

[6] D. Malkhi, M. Reiter and A. Wool, "The load and availability of Byzantine quorum systems," *SIAM Journal of Computing*, vol. 29, no. 6, pp. 1889-1901, April 2000.

[7] L. Lamport, R. Shostak and M Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 2, pp 382-401, 1982.

[8] F. Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM*, vol. 34, no. 2, pp. 56–78, 1991.

[9] Y. Amir, B. Coan, J. Kirsch and J.Lane, "Byzantine replication under attack," *Technical Report CNDS-2008-1*, Johns Hopkins University, www.dsn.jhu.edu, March 2008.

[10] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 447, November 2002.

[11] F. C. Gärtner, *Fundamentals in Fault Torelance*, Darmstadt Univ. of Technology, Germany.

[12] T. C. Bressoud and F. B. Schneider, "Hypervisor-Based Fault-Tolerance," *Isis Distributed Systems*, Cornell University.

Ioan PETRI has graduated the Faculty of Economics and Business Administration, Business Informatics dept from Babes-Bolyai University of Cluj-Napoca in 2007. On his doctoral orientation in Service Collaborative Systems develops a secured collaborative architecture that is expected to increase previous performances. He is interested in programming languages and databases as well as he particularly enjoy using new Java development frameworks.