2018 IEEE International Conference on Computational Science and Engineering

Analysis and Design of DOORS, in the context of Consistency, Availability, Partitioning and Latency

Dorin Palanciuc
Computer Science Department, Faculty of Automatic Control and Computers
***Politehnica** University*
Bucharest, Romania
dorin.palanciuc@gmail.com

***Abstract*—DOORS is a distributed system proposal that provides execution and storage services in the form of ”objects”, which encapsulate both state and behavior. We start by briefly describing the current state of the DOORS solution, as detailed in previous work. We then outline the class of problems that we aim to solve with DOORS, and provide brief motivations for the architectural choices. As direct consequences of the chosen architecture (distributed message passing in favor of shared memory) we analyze the following critical aspects: concurrency control, replication and the choice between consistency and availability. With the support of this analysis, we identify and present 3 different ”kinds” of inter-node communication. In spite of their apparent similarity, these scenarios are addressed differently into the design, such that the implementation of DOORS remains correct, consistent and useful.**

***Index Terms*—message passing, consistency, availability, latency, network partitioning, geographically distributed**

I. The Principles and overall architecture of DOORS

The subject is a distributed computing system which combines the functions of a distributed run-time environment with those of a distributed database, into a simple and useful abstraction. The present paper builds on the work described in [1], in which we argue that a unified architecture, covering both run-time and storage capabilities, in a limited distributed environment of tens-to-hundreds of nodes leads to a simpler, more robust and hopefully easier to understand and maintain solution for many distributed computing problems.

DOORS aims to implement the notion of object in the Smalltalk definition, albeit in a distributed environment:

- an object contains state and the behavior, in the forms of attributes and methods that operate on them;
- objects communicate by sending and receiving messages;
- the object that receives the message decides how to respond (typically by executing a method).

A. *Services and Node Interfaces*

The DOORS run-time must therefore provide services such that external entities - the system clients - may:

- define object structure and behavior;
- instantiate objects. Distribution of the instances across all available nodes must be handled by the run-time;
- allow entities to exchange messages. The details of reaching all targeted objects regardless of their location must also be handled by the run-time.

In [2] we elaborate on the overall architecture of DOORS, and we also mention the available interfaces and services that any DOORS node must provide:

1) Administrative Interface for managing the node lifecycle from an external, privileged role;
2) Client Interface for providing services to external entities . The relatively rich set of calls that we envision is meant to make DOORS a viable solution for real-world distributed problems;
3) Inter-node Interface connect individual nodes and realize the actual distributed storage and run-time environment.

B. *Communication sub-system*

All the above interfaces share the same low-level communication implementation, which is described in [2]

The major features of this communication solution are:

- event driven - asynchronous structure achieved by using **libevent**: [3];
- uses timeouts in propagating the events up, towards the node run-time; This also ensures progress in the absence of peer activity
- uses state machine constructs which ensure the propagation for only complete messages to the node run-time.

The most important function of the network interface is that it abstracts away the low level matters:

- that once a peer (or client) starts to send a message, it will either be fully received or completely ignored by the destination node;
- that if a peer closes the connection unexpectedly, the node receives a dedicated event, which is further used to define node behavior. We will not discuss the case of a client disconnecting, as this is an event that a DOORS node must handle gracefully without any side-effects upon node state. However, a peer closing the connection unexpectedly is a case of system partitioning, and this calls for substantial changes in the state of the node.

Each of those are are discussed in previous work, as indicated.

II. The problems at hand

The need for DOORS emerges from the analysis of a particular class of distributed problems, with the following characteristics:

- There are many, heterogeneous, geographically distributed data sources and data sinks, which produce/consume data asynchronously (e.g. telemetry devices, control equipment)
- The data sources have limited local resources, in terms of computing power, storage capacity, communication bandwidth. The employed communication channels may have low reliability.
- The data contained needs to be processed and aggregated into central repositories. Additional data articles are being produced as an effect of the aggregation, as well as outputs, that need to be routed back to the remote data sinks. The actual processing is performed on dedicated nodes, which are neither sources nor sinks, run in distinct locations and have radically different capabilities (e.g. dedicated servers running in data-centers, virtual machines running in cloud environments, or even cloud services running in PaaS setups)
- Some of the processing and outputs need to be elaborated with low latency, mandating for processing ”in the field” instead of ”at the headquarters”
- The size of the system is large enough such that long-term deployment, and successive functional refinements need to be implemented during ”production run-time”

Apart from the obvious examples of large telemetry and SCADA systems, we could also mention safety oriented, distributed applications such as railway signaling, water resource management, as well as non-critical, consumer oriented, IoT applications.

Due to the geographically distributed nature of the inputs and outputs, coupled with low latency requirements, such systems are less suitable for cluster computing. Furthermore, in order to be less sensitive to the performance variability of the available communication infrastructure, we choose to adhere to the message-passing paradigm instead of distributed shared memory. Communication and node failures become first-class events in DOORS, and are easily distinguishable and addressable by the solution implementation, instead of being hidden behind a generic ”distributed memory address range unavailable” exception. One could argue that we trade off performance for flexibility, which is reasonable when designing a general purpose system.

This trade-off becomes apparent when we take into consideration the recent performance improvements in the shared memory realm, based on firmware/hardware enhancements such as RDMA (Remote Direct Memory Access) and HTM(Hardware Transactional Memory). An interesting, RDMA based distributed transaction management system is proposed by [4]. An even higher performance level is claimed by [5], by employing hardware supported lock elision, albeit in cluster setups.

Moving towards cloud-based solutions, we evaluated [6], which proposes a cloud-based file-system for the enterprise. A BlueSky-based solution would indeed solve the storage aspect in a reliable way, however, it does not provide for run-time capabilities. At least for the moment, the major cloud suppliers have distinct offers for storage and computing (e.g. Amazon’s S3, complementing EC2)

Recent refinements in the cloud realm, in the forms of fog and edge computing are conceptually closer to the goals of DOORS, for decentralization of computing and shifting of control. An excellent introduction, albeit quite philosophical can be found in [7].

We can now conclude that DOORS fits best in the area of edge computing, observing that our focus is on mission critical, bespoke implementations rather than consumer-oriented, mobile computing. We continue the analysis and design effort on the main problems that emerge from the distributive nature of our solution: concurrency control (which needs to be solved even in the case of a single-node system, if the said node accepts multiple, simultaneous client connections), replication for redundancy, and consistency versus availability, especially in the event of system partitioning.

Given the complexity of these subjects in the context of practical implementations of distributed systems, we continue the research effort in a simplified version of the target architecture. We therefore propose the ”Minimal DOORS”, as further described.

III. Minimal System

DOORS development is at the beginning, and the basic architectural and functional features are being refined. For the purpose of clarity, in the current paper we choose to limit the scope of the system, such that:

1) a system instance must be able to host only objects with similar, and very simple structure;
2) every object contains:
    a) two attributes with integer values, *a* and *b*, which are directly accessible and modifiable (they are *public* in the traditional OOP interpretation);
    b) a method without side-effects, which when executed, returns the sum of the attributes;
    c) a method with side-effects, which receives two objects as parameters, and, if written in a semantic approximation of the Java language, has the following specification:

```
void m(Object y){
this.a=this.a+y.a;
this.b=this.b+y.b;
}
```

In the context of the above, a DOORS application is a specific kind of object which:

- is responsible of creating and initializing multiple instances of entities (in the order of thousands in this simplified version);

- is managing the life-cycle of all entities and provides access to them for external system clients.

IV. Concurrency Control

Moderation of concurrent control is needed in a system that allows simultaneous read and write access to a resource from more than one interface. This is allowed in DOORS even when the system is composed of a single node; several clients may connect to the node and send messages to the same object simultaneously. If more than one of these concurrent messages have successive side-effects on the object state, data integrity is affected. This is avoided in systems that provide serializability, such that a succession of data reads and/or writes are executed atomically, and at the end data remains in a state as if the operations were executed one after the other. A simple, albeit not the most efficient method to achieve serializability is two-phase locking. Most database systems of today (and yesterday) offer configurable levels of isolation between concurrent transactions, from ”read uncommitted” to ”strict serializable”. The default level is usually optimistic and leaves room for anomalies that affect data integrity - the most popular is a kind of multiversion concurrency control named ”snapshot isolation”, in which each transaction T sees the system state as produced by all transactions that were committed before T was started, and no effects are seen from transactions that are at least partially simultaneous with T.

Snapshot Isolation was formalized by Berenson et al in [8], and it was shown that non-serializable execution is possible. Also Cahill proposes a fully serializable version of the snapshot isolation in [9]

In the interest of simplicity, we choose to implement S2PL (Strict Two-Phase Locking) in DOORS, as described in [10], and disregard the impact of performance, at least in this reference implementation. The deadlock avoidance mechanism that is needed for a concrete S2PL implementation is the subject of a future article.

V. Distributed Transactions

A correct implementation of concurrency control provides DOORS with single-node transactions. If we make sure that any possible transaction in DOORS is limited to objects living on a single node, we could provide a fully ACID system at any scaling level. Furthermore, the actual number of nodes in the system could grow indefinitely without any effect on the transaction execution performance.

Real-life problems demonstrate otherwise. The simplest counter-example on the minimal system described in section III is a single transaction that creates a new object S, and then iterates through all the other instances in the system, sending the *m* message to S with each instance as parameter, effectively computing the sum of all entities in the system. Such a transaction is unavoidably distributed, and therefore not infinitely scalable. Specific mechanisms that reach consensus on whether to commit or rollback the distributed transaction need to be put in place.

A simple algorithm that ensures atomic commitment protocol for a distributed transaction is the Two-phase Commit [11]. The disadvantage that needs to be addressed in the concrete implementation is the inherent blocking nature due to the existence of a ”single point of failure”. Transaction coordinator (which, for simplicity could always be chosen as the node that originates the first message in the transaction) may fail, effectively stopping progress. DOORS can solve this, by relying on events and timeouts, both for individual messages and for entire transactions. It must be noted that, as stated by [11], two-phase commit does not guarantee fault tolerance. DOORS can afford to employ 2PC due to the fact that it considers fault tolerance as a separate aspect and proposes distinct mechanisms to address it. This is the next topic of the paper, and in order to approach it, we shall focus on the most frequent fault in a distributed system: network partitioning.

VI. PACELC and its influence on the behavior of a DOORS node

We are all familiar with the CAP theorem, formulated by Eric Brewer. It was first stated in 2000 at the Symposium on Principles of Distributed Computing and formally proven in 2002, by Nancy Lynch and Seth Gilbert. We are now also aware of the pitfall of considering it a classic trilemma (a choice of at most two options out of three). Further study shows that *Consistence*, *Availability* and *Partition Tolerance* have asymmetric roles [12]. In fact, all three are achieved, as long as partitioning of the system does not occur. In real life, the network keeping all the nodes together fails sometimes. When this happens, the system must choose between providing consistency or availability.

PACELC builds on the CAP theorem and shows that even in the absence of network partitioning, trade-offs need to be made: between consistency and latency. This is formalized in a paper published in 2012: [13], and the reason for this inevitable trade-off is the fact that a distributed system that ensures availability must always replicate data, and replication needs extra time (latency). In the words of Abadi: ”Ignoring consistency/latency trade-off of replicated systems is a major oversight in CAP, as it is present at all times during system operation, whereas CAP is only relevant in the arguably rare case of a network partition”. PACELC therefore stands for (”Partitioning then Availability or Consistency, Else Latency or Consistency”).

Databases that guarantee ACID will choose consistency over availability in the case of network partitioning (usually denoted as P+C). The most databases that are part of the NoSQL family, are designed around the BASE philosophy (Basically Available, Soft State, Eventual Consistency), and are suitable to problems in which system availability is more important that consistency of state such as e-commerce sites. These are usually P+A.

When we consider normal operation of a distributed database, without any partitioning, and we also impose that the system provides data replication, we notice that in order to guarantee consistency for every write, the system must first achieve consensus among all replicas. This adds non-negligible latency, until the system ensures that all copies are consistent. If on the other hand we choose to decrease latency and read the closest-available replica of the data, we increase the chance of obtaining an inconsistent read.

A brief analysis of the available distributed database systems reveals that there are two major popular approaches:

- The NoSQL approach: P+A and E+L (Availability in the presence of Partitioning, Else decrease Latency). Notable examples are: DynamoDB, Cassandra, CosmosDB, Riak;
- The NewSQL (ACID) approach: P+C and E+C (Consistence both in the presence and in the absence of Partitioning. Examples: VoltDB/H-Store, Megastore, MongoDB.

There is a special case that provides P+C and E+L, and this is PNUTS [14]. An interesting comment on this can be found on Daniels Abadi blog [15]. It is a definitely interesting trade-off chosen by the developers of PNUTS. The system is consistent in the presence of partitions, but fights to decrease the latency when network is operational, even when this means increasing the chance of inconsistencies.

VII. Message sending semantics in DOORS

We stated that DOORS uses the Smalltalk definition of objects, and that a DOORS system allows them to exchange messages. We choose the synchronous semantics for message sending in DOORS. This means that:

- any client C that sends a message to an object in a DOORS system will receive an answer;
- the answer must be available to C in a predefined time-frame. If no timely answer is received, the client may safely assume that the message produced no change of state into the system;
- any side-effects of the message being received occur before the answer is available.

Asynchronous semantics for message sending in DOORS, in which any of the above are not true, may be subject of a future version of DOORS.

A. Why do nodes communicate in a distributed system?

We identified three main reasons for which we perform inter-node communication in a distributed system:

- sending messages (we trigger code execution on the destination node);
- redundancy: we copy objects on a nearby node so that data is not lost and also processing may continue in the case that our original node fails;
- object movement or copying: we copy objects on a faraway node because they need to be available to "someone else", which is close to that faraway node. This operation has relatively higher latency and also chances of connection loss.

We propose that DOORS serve all the three cases of inter-node communication and below we describe the chosen system behavior for each case.

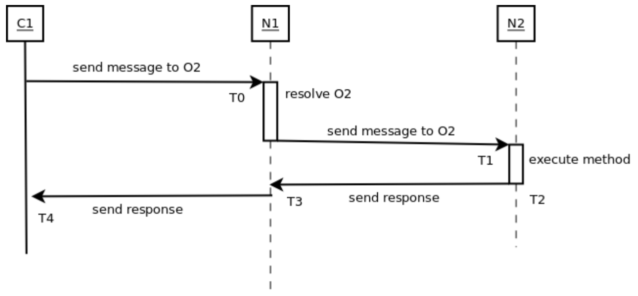

Fig. 1. Sequence of event in the case of messaging for scaling

B. Message sending

The simplest and most frequent form of inter-node communication, it has the following crucial aspects: Messages are simple and ephemeral objects. They are born when being sent, and disappear when they are being received. Of course, they hold state, and may be temporarily queued, but most importantly, messages are immutable. Scaling of the system is achieved by sending messages, they may trigger changes of state and execution of methods.

Let there be C1 a client of the system, N1 the node to which the client is connected directly (via the client interface) and N2 the node which hosts O2, the object targeted by the message sent by the client.

The communication that occurs in the system is depicted in Figure 1

We denoted the important moments with Tx where:

- T0 N1 receives the request from the client;
- T1 N2 receives the request from N1, identifies the method to be executed as a response to the message and starts executing it;
- T2 Execution concludes on N2 and the result is being returned to N1;
- T3 Result is available on the N1;
- T4 result is available to the client.

A connection loss between C1 and N1 means that the transaction (T0 T4) fails immediately. A connection loss between N1 and N2 means system partitioning. Since all objects have a single copy, O2 becomes unavailable and the transaction fails. Due to the synchronous semantics, a rollback is needed on N2 if partitioning occurs between T1 and T2, when the execution produced its side-effects.

C. Inter-node communication for redundancy.

DOORS will provide a system-wide, configurable replication factor. A value of 1 means no replication. A value of 2 means that the system will maintain two copies for each object, such that loss of a node will not lead to loss of data. A consensus mechanism must be provided, so that all replicas are maintained in sync. The code, however, even if present on both replicas, will be executed only once, and therefore, a master copy is defined for each object, and the node that hosts the master copy will also execute the code. All state

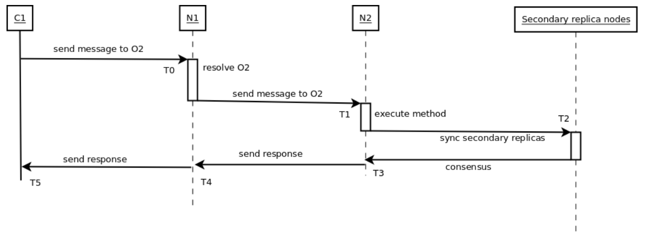

Fig. 2. Sequence of event in the case of messaging for redundancy

changes occurred as a result of the execution (side-effects) will be synchronized to the secondary copies. We choose DOORS to be E+C, which means that, in the absence of partitioning, consistency is chosen over low-latency. This translates to the following behavior:

- the response to any message is postponed until all object replicas are in sync;
- in the case of partitioning between the master and any secondary replica, the response is postponed until a new complete set of replicas is built and synchronized.

Let there be two nodes, N1 and N2, in a DOORS system that contains also several other nodes. N1 has a client connected, C1, which sends a message to an object O2, that has its master copy on node N2 and other copies spread around the system (there might even be a secondary copy on N1, but it will never be directly targeted by C1). The communication that occurs in the system is depicted in Figure 2

We denoted the important moments with Tx where:

- T0 N1 receives the request from the client;
- T1 N2 receives the request from N1;
- T2 Execution concluded on the master replica and synchronization of secondary replicas begins;
- T3 Consensus on the replica synchronization has been achieved;
- T4 Result is available on the N1;
- T5 result is available to the client.

There are two kinds of partitioning in the scenario above:

1) Loss of connection between the master replica and any of the secondary replicas. This must not be visible to the client. The system can be configured to either ignore this or create new secondary replicas on other available nodes. If the system chooses Availability, the result of the execution will be immediately returned (T2 = T3). DOORS chooses Consistency, so the effect is a delay of T3 until the sync consensus is reached;
2) Loss of connection between the N1 (the entry point of the request) and the master replica N2. This is always visible to the client, as it is a loss of connection between C1 and N2. For this case, it is relevant to specify system behavior based on the moment when the partitioning has been detected by N1. If the partitioning is detected in the interval T0 - T1 then the system must construct another master replica (based on secondary replicas). If at least one secondary replica is accessible to N1 (let us assume that it is hosted on N3), then N3 becomes the new master replica. For systems that choose Availability, execution can start on N3 and the result is being returned as soon as it is available (T2=T3). On DOORS, we choose Consistency, so T1 is delayed until the needed replica set is constructed, and T3 is delayed until all new replicas are in sync. Of course, if no secondary replica is accessible from N1, then execution fails and the failure is returned to the client.

If the partitioning is detected in the interval T1 T4 then the system must rollback all side-effects.

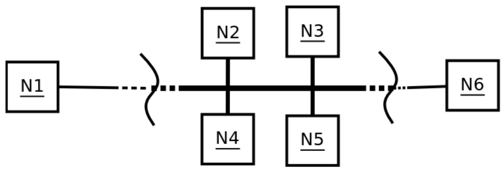

Fig. 3. Object movement over long connections

D. Inter-node communication for object movement or copying

This is the case in which objects need to ”move” geographically. One crucial aspect is that, when an object O moves from N1 to N2, it means that, by the nature of the problem, the next messages (and in consequence the changes in state) are most probably to originate in N2 or at least in other nodes that are closer (read ”have lower latency and less prone to partition”) to N2. The object movement scenario is used extensively in the cases when, due to infrastructure topology and geographical dimensions, the partitioning is more likely to appear between N1 and N2 and latency is relevant with regard to the duration of the possible transactions. It shall be noted that performing an object copy from N1 to N2 is done in order to make its state available on N2, and not in order to provide redundancy.

Here are two examples of communication scenarios that fit the above:

1) Data transmission from geographically distributed nodes to a central repository in a telemetry or IoT application. Here the data is acquired by faraway nodes, and then sent over non-reliable connections to a center node, which has the role of data aggregator (and main processor). The DOORS service that is used by the remote nodes in order to send the acquired data home is a different service than the one used for example to keep several replicas of this data on a set of nodes, all of them present in the central location, as it can be seen in Figure 3.
2) A cash withdrawal operation, in which the cash machine must first request the current status of the account (out of which, the balance attribute is the most relevant) from the bank (the central node), and then, if the case, perform the actual transaction.

Let there be two nodes, N1 and N2, in a DOORS system that contains also several other nodes. N1 has a client connected C1. C1 sends a message to an object O2 which is known to

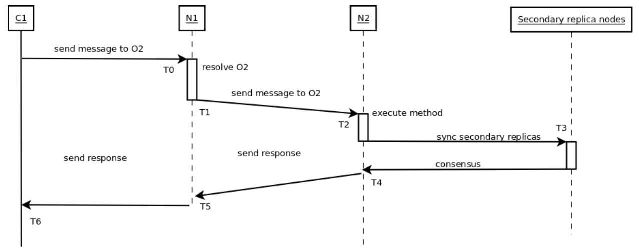

Fig. 4. Sequence of event in the case of messaging for communication

be remote from N1. The master copy of O2 is on N2. The communication that occurs in the system is depicted in Figure 4

The scenario above is mostly similar to the redundancy scenario depicted in figure 2, with the observation that the time intervals T1-T2 and T4-T5 are not negligible and that partitioning is more probable.

Loss of communication between N2 and the secondary replica nodes is detected and processed identically with the redundancy case, described in section VII-C. The difference appears in the case of partitioning between N1 and N2. As the N1-N2 channel is less reliable and prone to higher latency, the system will increase the timeouts and also perform multiple attempts to send the message to N2. Eventually, it will to elect a new master and build a new complete replica set, but much later than in the redundancy case. The interval T2-T4 is processed in similar manner with the redundancy case, as it is assumed that the latency between N2 and its secondary replica nodes is significantly lower than the latency between N1 and N2. If partitioning is detected in the interval T4-T5, the system will become more permissive again in order to fulfill the increased latency imposed by the N1-N2 connection. It must be noted that the transaction starts at T0 and lasts until T5. Due to the consistency constraint, the object targeted by the message will be locked until T5. In a similar manner with the redundancy scenario, if partitioning occurs in T1-T4, a rollback of the message side-effects will be performed, however, the timeout is significantly more permissive (for some applications, T1-T4 may span several days!). Inter-node communication over large distances has message queuing behavior and concurrency issues are significantly more probable (during the several days that are needed for the message to reach N2 and then the answer to reach N1, it is highly likely that other nodes send other messages to the same object on N2. The message processing turns into a concurrency control problem and the system will be eventually consistent if any emerged conflicts are successfully resolved. As stated in [16], in the presence of high latency or unreliable channels, partition recovery becomes a problem of conflict resolution. For this, the use of Conflict-Free Replicated Data Types [17], as well as Operational Transformations [18] are possible solutions.

VIII. Conclusion and future work

The paper restates the class of problems for which DOORS is being built, it refines the concept and presents the major distributive features and design decisions on which the concrete implementation relies. In the analysis and presentation of these, we presented a simplification of DOORS, such that the system behavior in relevant distributive scenarios is easier to describe and validate. In the end, we identified three distinct goals for inter-node communication, with subtle but crucial conceptual differences - the immutable and ephemeral character of messages, the ”atomic appearance” of replication and the essentially different semantic of communication when objects are being moved or copied. In the future, we propose to analyze the systems in which partitioning is not a fault, but a normal state, in which the system may spend long periods of time (e.g. collaborative editing of documents). These systems aim for improving latency and delay the achievement of consistency. Reconciliation of potentially divergent copies of the same object is a non-trivial problem for which Consistent Replicated Data Types and Operational Transformations are possible solutions.

References

[1] D. Palanciuc, “DOORS: Distributed object oriented runtime system (position paper),” in *2017 16th International Symposium on Parallel and Distributed Computing (ISPDC)*. Innsbruck, Austria: IEEE, jul 2017. [Online]. Available: http://dx.doi.org/10.1109/ISPDC.2017.20

[2] ——, “The interfaces of a DOORS node - proposal for an event-driven architecture for node communication. reference implementation.” POLITEHNICA University of Bucharest, Tech. Rep. 3, February 2018.

[3] N. Mathewson, A. Khuzhin, and N. Provos, “Libevent - an event notification library,” http://libevent.org, cited June 2018.

[4] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, “No compromises,” in *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP '15*. ACM Press, 2015. [Online]. Available: https://doi.org/10.1145/2815400.2815425

[5] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, “Fast in-memory transaction processing using RDMA and HTM,” in *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP '15*. ACM Press, 2015. [Online]. Available: https://doi.org/10.1145/2815400.2815419

[6] M. Vrable, S. Savage, and G. M. Voelker, “Bluesky: a cloud-backed file system for the enterprise,” in *Proceedings of the 10th USENIX Conference on File and Storage Technologies - FAST*. Association for Computing Machinery (ACM), 2012. [Online]. Available: http://www.usenix.net/legacy/events/fast12/tech/full_papers/Vrable.pdf

[7] P. G. Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, “Edge-centric computing,” *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 5, pp. 37–42, sep 2015. [Online]. Available: https://doi.org/10.1145/2831347.2831354

[8] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, “A critique of ANSI SQL isolation levels,” in *Proceedings of the 1995 ACM SIGMOD international conference on Management of data - SIGMOD '95*. ACM Press, 1995. [Online]. Available: https://doi.org/10.1145/223784.223785

[9] M. J. Cahill, U. Röhm, and A. D. Fekete, “Serializable isolation for snapshot databases,” *ACM Transactions on Database Systems*, vol. 34, no. 4, pp. 1–42, dec 2009. [Online]. Available: https://doi.org/10.1145/1620585.1620587

[10] P. A. Bernstein and N. Goodman, “Concurrency control in distributed database systems,” *ACM Computing Surveys*, vol. 13, no. 2, pp. 185–221, jun 1981. [Online]. Available: https://doi.org/10.1145/356842.356846

[11] I. Keidar and D. Dolev, “Increasing the resilience of atomic commit, at no additional cost,” in *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems - PODS '95*. ACM Press, 1995. [Online]. Available: https://doi.org/10.1145/212433.212468

[12] E. A. Brewer, “Towards robust distributed systems (abstract),” in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC '00*. ACM Press, 2000. [Online]. Available: https://doi.org/10.1145/343477.343502

[13] D. Abadi, “Consistency tradeoffs in modern distributed database system design: CAP is only part of the story,” *Computer*, vol. 45, no. 2, pp. 37–42, feb 2012. [Online]. Available: https://doi.org/10.1109/mc.2012.33

[14] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, “PNUTS,” *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, aug 2008. [Online]. Available: https://doi.org/10.14778/1454159.1454167

[15] D. Abadi, “Problems with CAP, and Yahoo’s little known NoSQL System,” http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html, cited June 2018.

[16] E. Brewer, “CAP twelve years later: How the ”rules” have changed,” *Computer*, vol. 45, no. 2, pp. 23–29, feb 2012. [Online]. Available: https://doi.org/10.1109/mc.2012.37

[17] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “Conflict-free replicated data types,” in *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 386–400. [Online]. Available: https://doi.org/10.1007/978-3-642-24550-3_29

[18] C. A. Ellis and S. J. Gibbs, “Concurrency control in groupware systems,” *ACM SIGMOD Record*, vol. 18, no. 2, pp. 399–407, jun 1989. [Online]. Available: https://doi.org/10.1145/66926.66963