Master research Internship

Bibliographic report

Enforcing Syntactic and Semantic Integrity in Distributed Systems

Domain: Distributed, Parallel, and Cluster Computing - Data Structures and Algorithms

Author:
Grégoire Bonin

Supervisor:
Achour Mostéfaoui
Hala Skaf
GDD - LS2N

Abstract: This bibliographical report talks about the different consistency criteria in distributed systems, more specially the update consistency and the integration of integrity constraints to provide syntactic and semantic consistency. The goal of the internship will be to find a way to combine both these criteria to have a stronger criteria by enforcing both syntactic and semantic integrity in distributed systems at the same time.

Contents

1 Introduction

During this internship, I combine a consistency criterion with integrity constraints to guarantee more properties on distributed objects.
A distributed system is composed of several nodes communicating together and sharing memory [5]. Consistency criteria in distributed systems are used to describe objects to indicate which properties they satisfy and what assumption are required to use them in distributed systems. Consistency criteria can sometimes be compared one to another in terms of strength : the stronger the criterion, the stronger the properties it satisfies are, as exposed in [7]. These properties are often the possibility to assert an order (partial or global) on the operations made by distributed applications in a distributed system : this is a syntactic property as there can be semantic issues depending on the order found. Here we consider the Update Consistency [8] which is a refinement of the Eventual Consistency but remains a "weak" criterion when compared to linearizability. Finding new consistency criteria is an important subject as they can bring better answer to some problems (more flexible/resilient data structures for distributed applications).
On the other hand, in databases research focus on preserving the semantic integrity of the data in the different nodes : this is done by enforcing integrity constraints [11]. This semantic criterion strongly depends on the problem we are trying to solve in opposite to the consistency criteria which are global.
The goal is to find a way to combine both Update Consistency with the enforcement of Integrity Constraints to be able to have the two advantages of having semantic and syntactic integrity while keeping the new criterion independent of applications.

This report is organized as follows. Section 2 presents the Update Consistency and its properties based on several articles. Section 3 details the integration of Integrity Constraints in distributed nodes. Section 4 describes in details the goal,*i.e.*, combining both Update Consistency with Integrity Constraints. The last section concludes this report.

2 A syntactic point of view

Update Consistency is a refinement from the eventual consistency criterion [8]. Eventual Consistency (EC) states that, in a distributed system, when all processes stop updating an object, it will eventual converge towards a unique state [12].
The issue with Eventual Consistency is that before converging, the different nodes don't have consistent states.
Update consistency aims at making a stronger eventual consistency : Update Consistency ensures that all nodes converge towards a unique final state after updates are finished, but it also ensures that all the states of the distributed object are consistent with a linearization of the updates (this means that it is a linearization of "write" actions only, as "reads" do not modify the object).
This is weaker than Linearizability or Sequential Consistency, but stronger than Eventual Consistency.

In the following sections, we will present the model in which we are working, then we will briefly present Eventual Consistency, to finally study the criterion we are interested in : Update Consistency.

2.1 System's Model

We are working in wait-free systems, which means asynchronous systems where all but one process may crash. These systems are described in [5]
The objects we are considering are UQ-ADT (Update Query Abstract Data Type) Here is the definition given in [8] : An UQ-ADT is consisting of a t-uple $(U, Q_i, Q_o, S, s_0, T, G)$ where :

- U is the set of update operations (countable)
- Q_i and Q_o are the input and output alphabets (countable as well). $Q = Q_i \times Q_o$ is the set of query operations where the couple (q_i, q_o) is a query operation where query q_i returns the value q_o, denoted q_i/q_o
- S is the set of states (countable)
- $s_0 \in S$ is the initial state
- $T : S \times U \to S$ is the transition function
- $G : S \times Q_i \to Q_o$ is the output function.

This is called the specification of an object. A sequence of update operations $(o_n)_{n\in\mathbb{N}} \in (U \cup Q)^t$ is said to be recognized by such a specification if :
$\exists (s_i)_{i\geq 1} \in S^t$ such that :

- $\forall i, T(s_i, o_i) = s_{i+1}$ if o_i is an update $(\in U)$ or $s_i = s_{i+1}$
- $G(s_i, q_i) = q_o$ if o_i is the query q_i/q_o

All the sequences recognized by the specification of O are denoted as $L(O)$.
After defining the specification of an object, we can give the definition of a distributed history of an object. This will then permit us to give a formal definition of consistency criteria.

A distributed history of an object is a t-uple $H = (U, Q, E, \Lambda, \mapsto)$ where :

- U and Q are respectively the sets of updates and queries
- E is the set of events (countable)
- $\Lambda : E \to U \cup Q$ is the labelling function
- $\mapsto \subset E \times E$ is the program order, a partial order such that : $\forall e \in E, \{e' \in E : e' \mapsto e\}$ is finite

$U_H = \{e \in E : \Lambda(e) \in U\}$ is the set of update events of H, and $Q_H = \{e \in E : \Lambda(e) \in Q\}$ is the set of query events of H.
A projection of the set of events $F \subset E$ on the history H, denoted $H_F = (U, Q, F, \Lambda, \mapsto \cap(F \times F))$ is the history H with the events of F only. Another projection on the history Hcan be done with a partial order $\to$ that respects the definition of a program order $\mapsto$, and it is denoted $H^{\to} = (U, Q, E, \Lambda, \to \cap(E \times E))$. it consists of the history H with the events ordered by $\to$.

Now we can characterize formally a consistency criterion : Here is the definition given in [8].

A consistency criterion C is a function that, given a UQ-ADT O, returns the set of all possibles histories that O permits, denoted $C(O)$. For a shared object SO, instance of O, SO is C-consistent if all histories that can be created with SO are in $C(O)$.

With this definition, we can give a precise definition of eventual consistency, and update consistency.

2.2 Eventual Consistency

Eventual Consistency was introduced in 2008 by W.Vogels [12]. This property ensures that once every node or process stops to update an object that is eventually consistent, the returned values of "read" operations will converge towards a unique value. it is important to note that if the nodes never stop updating, the property is still satisfied.
The time between the last update and the convergence state is called the inconsistency window. During this window, the different states of the different processes are inconsistent : different processes trying to read in the same memory won't necessary return the same values, which can be a problem.
As W.Vogels says in [12] the duration of this inconsistency window can be estimated from the parameters of the system : communications delay, number of nodes, system topology ...

Such a consistency criterion is said to provide weak consistency. When dealing with strong consistency criteria such as Linearizability or Sequential Consistency, every state is consistent and all read/write operation will return the same value in every process as there is a total order on the operation, which is a really strong property, but it comes with a enormous drawback : the loss of great availability needed in distributed application : in order to provide strong consistency, operations can take a long time which restrains availability as a process has to wait for the operation to spread to every node before being able to executes its own operation (otherwise, the sates wouldn't be consistent).

This is where Eventual Consistency becomes usefull : it permits to keep good availability while keeping a consistency property.
This property however, is not strong enough as everything can happen during the inconsistency window.

Thanks to the definition of consistency criterion given previously, we can give a formal definition of Eventual Consistency :

A history H is said to be eventually consistent if U_H is infinite (processes never stop updating) or $\exists s \in S$ such that $\{q_i/q_o \in Q_H : G(s, q_i) \neq q_o\}$ is finite (the number of queries that return non consistent values in state s is finite).

Along with Eventual Consistency, there is another consistency criterion, called Strong Eventual Consistency, which differs from Eventual Consistency in the sense that the convergence of several replicas in different nodes happens as soon as the same updates are received. the notion of replicas and message reception are captured by a visibility relation $\xrightarrow{vis}$. Strong Eventual Consistency is then defined as follows :

A history H is strong eventually consistent if there is a relation $\xrightarrow{vis}$ that is acyclic and reflexive, and such that it satisfies the 3 following properties :

- Eventual delivery : an update will eventually be visible by all the nodes : that is to say $\forall u \in U_H, \{e \in E, u \overset{vis}{\nrightarrow} e\}$ is finite
- Growth : once an event has been viewed, it remains visible, that is to say $\forall e, e', e" \in E, (e \overset{vis}{\longrightarrow} e' \wedge e' \mapsto e") \Rightarrow (e \overset{vis}{\longrightarrow} e")$
- Strong convergence : $\forall V \subset U_H, \exists s \in S, \forall q_i/q_o \in Q_H, V = \{u \in U_H : u \overset{vis}{\longrightarrow} q_i/q_o\} \Rightarrow G(s, q_i) = q_o$ which means that two queries operations can be issued in the same state if the have the same update past.

We will now study a stronger weak consistency criterion : Update Consistency.

2.3 Update Consistency

Now that we have seen how to define consistency criteria and what are Eventual Consistency and Strong Eventual Consistency , we can focus on stronger consistency criteria : Update Consistency and Strong Update Consistency, introduced in [8].

Update Consistency and Strong Update Consistency are more relevant than Eventual consistencies in the sense that they impose constraints on the states in the inconsistency window. Eventual Consistency and Strong Eventual Consistency do not consider the sequential specification of the shared objects, thus some implementation can be (Strong) Eventually Consistent without respecting the object itself (an implementation ignoring updates for example).

In Update Consistency, in order to consider the sequential specification of an object, the updates have to satisfy a total order (hence the name update consistency) that contains the program order of the object $\mapsto$. This is still weaker than Sequential Consistency as in Sequential Consistency, the total order concerns both updates and queries. By noticing that, we can give another explanation of Update Consistency : if there is a finite number of updates, then it is possible to remove a finite number of queries to render the history sequentially consistent.

Now that we've explain the concept behind Update Consistency, we can give the formal definition of it :

A history H is update consistent if the processes never stop updating : U_H is finite, or if it is possible to remove a finite set of queries to render the history linearizable : $\exists Q' \subset Q_H$ such that $lin(H_{E \setminus Q'}) \cap L(O) \neq \emptyset$ where $lin(H)$ correspond to the set of all sequential history of H with the same events, and respecting the program order of O.

In the same fashion as Strong Eventual Consistency was added to Eventual Consistency, Strong Update Consistency was added to Update Consistency. As previously, it relies on the visibility relation $\overset{vis}{\longrightarrow}$, but it adds another requisite : a total order $\leq$ that contains $\overset{vis}{\longrightarrow}$. These relations have to satisfy the 3 following properties, the first two being the same as for Strong Eventual Consistency :

- Eventual delivery : an update will eventually be visible by all the nodes : that is to say $\forall u \in U_H, \{e \in E, u \overset{vis}{\nrightarrow} e\}$ is finite
- Growth : once an event has been viewed, it remains visible, that is to say $\forall e, e', e" \in E, (e \overset{vis}{\longrightarrow} e' \wedge e' \mapsto e") \Rightarrow (e \overset{vis}{\longrightarrow} e")$

- Strong sequential convergence : the result of a query depends on the updates it views, defined by the following set $\forall u \in U_H, lin(H^{\leq}_{V(q)\cup\{q\}}) \cap L(O) \neq \emptyset$ where $V(q) = \{u \in U_H : u \xrightarrow{vis} q\}$ is the set of updates visible by q, and $H^{\leq}_{V(q)\cup\{q\}}$ is the composition of the two projection of histories

In next section, we will detail how to maintain semantic consistency.

3 A semantic point of view

As we have seen previously, consistency criteria permit to ensure some consistency on the order of the operations, but no concerns are made about the semantic integrity of the program.
This was resolved in databases by adding integrity constraints and enforcing them when updates are made to the data [4], but the consistency management in databases is integrated with serializability as correction criterion, however it is a strong criterion.
This was also solved in a system implementing Operational Transformations (used for cooperative editing), permitting to enforce convergence, causality, and intention preservation [11]. The principle behind these integrity constraints is that the constraints are defined prior to deploying a distributed application, and they are enforced while replicating an action to all the nodes.

We will now look at how integrity constraints are defined, how and when they are checked, and how to respond to a constraint violation.

3.1 Constraints Definition

Integrity Constraints can be defined in two ways according to [11] : inside the application operation directly which implies no further checking whatsoever, or appart from the operation, as a set of constraints declaratively defined.
The first solution seems better as we do not have to check if constraints are satisfied at each operation, but it is way harder to define or modify them, as for each constraint, all operation have to be modified accordingly and proved formally. We will concentrate on the second way of defining the integrity constraints.

As there is no central node, the set of constraints will have to be replicated on each node and will become a shared object itself.

3.2 Constraints Checking

Here we have the questions of when ? and how ?

If we check if the constraints are enforced too often, this will slow the system, and if we check the constraints not often enough, this might lead to inconsistent states, so we have to find the good moment to check the constraints.
There are three possibilities concerning the when question:

- at the end of a local operation
- at the end of a local macro operation

- at the end of the integration of a remote macro operation.

If we check the integrity constraints at the end of each elementary operation, we will slow the system by a lot so this is not a good option.
A macro operation is a group of elementary operations, its result being the result of the sequence of the elementary operations that compose the macro operation. Macro operation are the ones to be broadcasted to keep all replicas the same. The semantic constraints might be violated during the macro operation so we need to check them after every macro operation. If the constraints are not checked at the end of a local macro operation, then semantic might be violated and propagated to other nodes.
Because the semantic constraints were respected after a node did a local macro operation does not mean that every node will satisfy the constraints once the operation will be replicated, so we also need to check the constraints after a remote macro operation has been integrated.

In order to enforce integrity constraints, several methods can be applied, as discussed more in details in [4] :

- Static analysis of the possible updates
- Modification of potentially violating queries
- Temporary workspace to analyse if an update is dangerous

3.3 Response to Constraints Violation

After a constraint violation has been detected, there are two possibilities :

- abort the current operation if possible
- repair the data to a consistent state.

The abort approach can be used in the case of a system with triggers, or buffer workspace when the update has not been made to the real data [4].

Repair operation are discussed more in details in [6].

Most approaches propose abort approach. we want to propose a general repair approach that integrates with Update Consistency independently of applications.

4 Why both Syntactic and Semantic Consistency ?

This is the core of the subject : why do we want to combine Update Consistency with Integrity Constraints to enforce both syntactic and semantic integrity ?

This work is the sequel of a previous internship by D.Maussion in GDD team, and we will present a motivating example in order to present the issue we are facing.

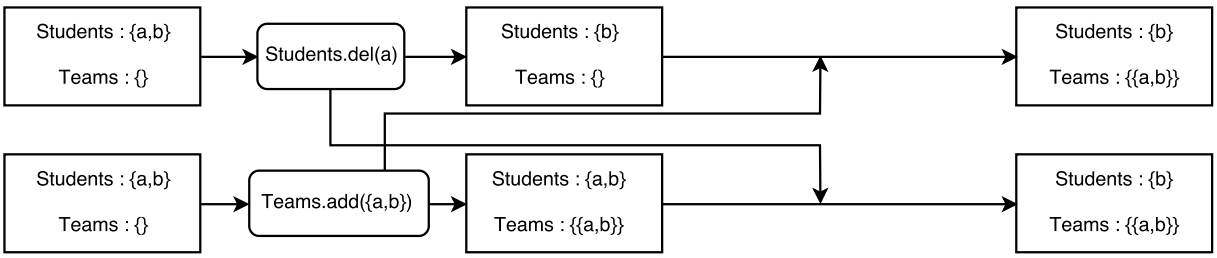

Figure 1: example of a semantically wrong convergence

Here we have two sets : Students and Teams. Semantically, we want that each member of a team appears in the Students set, this is our integrity constraint. In this example, both actions are respecting the integrity constraint when they are done ($Students.del(a)$ and $Teams.add(a, b)$) but the result after the convergence (thanks to Update Consistency) is not. This is why we have to find a consistency criterion that fulfils both roles : ensuring Update Consistence, and Respect to the Integrity Constraints.

In [11] a similar approach was presented but this is only defined for Operational Transformation systems, and not UQ-ADT.

In [3], in order to satisfy some constraint that are considered more important, the consistency criterion is changed for some operations (an example of a bank account requiring strong consistency is given). This is another approach to constraints enforcement.

In [10], M.Shapiro et al introduced Explicit Consistency that is defined in terms of application properties not like other consistency criteria, which are defined in terms of execution order.

5 Conclusion

This report presents syntactic consistency criteria and more precisely Update Consistency, and semantic consistency with Integrity Constraints. Update Consistency is a weak consistency criterion, which is a refinement of Eventual Consistency. Integrity Constraints are usually used in databases with strong consistency criteria to maintain semantic consistency. Our future work is to combine Integrity Constraints with Update Consistency. We aim to propose an approach to repair integrity violation while maintaining Update Consistency. This problem is challenging because it requires to define a new abstract type for repair operations compatible with Update Consistency.

References

[1] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, page 6. ACM, 2015.

[2] Neil Conway, William R Marczak, Peter Alvaro, Joseph M Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 1. ACM, 2012.

[3] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. *ACM SIGPLAN Notices*, 51(1):371–384, 2016.

[4] Paul WPJ Grefen and Peter MG Apers. Integrity control in relational database systemsan overview. *Data & Knowledge Engineering*, 10(2):187–223, 1993.

[5] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[6] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Consistency management with repair actions. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 455–464. IEEE, 2003.

[7] Matthieu Perrin. Modélisation des types de données abstraits dans les systèmes répartis à sémantique faible. 2016.

[8] Matthieu Perrin, Achour Mostefaoui, and Claude Jard. Update consistency for wait-free concurrent objects. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 219–228. IEEE, 2015.

[9] Matthieu Perrin, Achour Mostefaoui, and Claude Jard. Causal consistency: Beyond memory. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 26. ACM, 2016.

[10] Marc Shapiro and Mahsa Najafzadeh. Cise safety tool. 2015.

[11] Hala Skaf-Molli, Pascal Molli, and Gérald Oster. Semantic consistency for collaborative systems. In *Proceedings of the International Workshop on Collaborative Editing Systems-CEW*, 2003.

[12] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.