STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIV**, Number 2, 2009

GENERALIZED FORMAL DEFINITION OF CONFLICT DETECTION AND RESOLUTION

CIPRIAN COSTA

Abstract. In a distributed environment where information that is not read-only is shared between multiple parts of the system, the problem of conflict detection and resolution has to be addressed. In this paper we discuss the issue of generalized conflict definition because this main aspect of the consistency of any distributed system is often treated on a case by case basis. Based on the formal definition we can move on and identify what are the main types of conflict resolution algorithms which is very important for an accurate description of the guarantees offered by a distributed system.

1. Introduction

A lot of effort was channelled into making sure that the conflicts do not appear in a distributed environment mainly because a system that is fully consistent is a simple abstraction and we can reuse many of the concepts that were developed and proven in non-distributed systems. Several types of distributed systems have abandoned this line of thought and moved towards more complicated abstractions, where conflicts are allowed to exist. A simple example are classical desktop database front end applications that have moved from connected environments using pessimistic concurrency to disconnected environments using optimistic concurrency techniques in order to manage the relatively rare but possible conflicts. Why and when these types of systems are created and how to decide which to use is not in the scope of this paper. However, since they gained a lot of traction in recent times, we think that the notion of conflict detection and resolution deserves a unified and formal definition that can be used to better understand these distributed environments.

It is proven by the CAP theorem that availability, consistency and partition tolerance can not be achieved at the same time in a distributed system([2]). Consistency is a desirable property of any system since it makes programming and reasoning on the results of that system simpler, practically eliminating

Received by the editors: December 1, 2009.
2010 *Mathematics Subject Classification.* 68P15, 68M14.
1998 *CR Categories and Descriptors.* H.2.4 [**Systems**]: Subtopic – *Distributed databases*;
Key words and phrases. distributed systems, conflict.

uncertainty. In order to achieve consistency in a distributed system, a majority of the parts need to agree on every operation that affects the common state. In order to achieve that, a lot of consensus algorithms like Paxos ([6]) were created and found their way into various real live implementations like the Chubby system at Google ([3]).

The problem with consistency is that, according to the CAP theorem, we need to give up on availability or partition tolerance. Since in many distributed systems of large scale such requirements are non-negotiable because of the impact they would have on the perceived quality of the system, new approaches in which conflicts are allowed to temporarily exist in the system have been proposed ([10]).

Because the system is no longer consistent, it is much more difficult to build something on top of it (one may never be sure that the values returned by a part of the system are actually correct), it is important to define what type of guarantees can the system provide. We argue that an important part of this is generated by the way conflicts are detected and resolved. Throughout this paper we analyse existing work in the domain of conflict detection and resolution and propose a generalized conflict detection definition and some criteria that can be used for conflict resolution categorization.

2. Conflict definition

Informally, we define conflicts as situations in which one or more parts of a system have different representations of a shared fact.

Significant research on conflicts has been done in the field of autonomous agents, specifically addressing the problem of distributed constraint satisfaction DCSP ([11]). A constraint satisfaction problem is defined as finding an assignment to a set of variables with the property that a set of given predicates are satisfied by the said assignment. A DCSP is a CSP (constraint satisfaction problem) where the variables and the predicates are distributed among several independent agents and gathering all the information required for solving the CSP on a single node is impossible for various reasons (availability requirements, software incompatibility, etc). In this case, a conflict is defined as an assignment chosen by one of the agents that is valid under all the local predicates but does not hold in at least one of the other predicates.

Another field in which conflicts and conflict resolution algorithms were researched is collaborative editing ([8], [1], [5]). Several aspects related to conflicts and conflict resolution appear in collaborative editing that are not present in DCSPs:

- *Intent* - In the case of distributed agents the software that runs on each agent is well understood and the intent of any operation is always known. In the case of collaborative editing the intent of the user has to be inferred from their actions, it is not a priori known .

- *Operational transformations* - certain conflicts can be transformed in ways that allow them to be applied to the same document and preserve the operations of both users. This situation occurs when a user inserts a character at position 2 while another will insert a character at position 6. If we transform the operation of the second in an insert on position 7, both users can save their edits. This conflict is called a *non-exclusive conflict*. Extensive studies of operational transformations in various scenarios can be found in Sun and all 1998 [7] and Sun and all 2004 [9].

A definition that is often used in these scenarios is based on causality relations because they are supposed to capture the intent of the user. The supposition is that everything the user knows about a document is the cause of the intent, and any change in the cause could possibly alter the intent. From this, if two users are performing an operation having different knowledge of the environment, their actions could be in conflict. Informally, we say that $o_1 \to o_2$ (o_1 causally precedes o_2) if the user performing o_2 was aware of o_1 before performing o_2. A conflict in this case is defined as a pair of operations with the property that $o_1 \nrightarrow o_2$ and $o_2 \nrightarrow o_1$.

We consider this to be a pessimistic intent preservation. The problem with it is that, in some systems, it could lead to a lot of false positives and the main assumption of these distributed systems is that conflicts are rare and far between. We argue that we need a more generic definition of conflict that would allow a finer grained control over what constitutes intent, what part of the existing state is relevant for an operation and what alternatives are available to merge the two operations thus avoiding a conflict that might lead to an expensive negotiation in order to be resolved.

3. Generalized conflict definition

Let us assume that $x_1 \in D_1$, $x_2 \in D_2$, ..., $x_n \in D_n$ is a set of facts shared between multiple parts of the system. We define an operation as being the tuple

$$o_k = (x_{k_1}, ..., x_{k_p}, state_k, alt_k)$$

where k is the system that executed the operation, x_{k_1}, ... x_{k_p} are assignment on a subset of shared facts, *state* is a representation of the facts and other information that captures the relevant context of the operation, thus defining the intent of the user performing the operation, *alt* is a function defined as $alt : O \times O \to O \cup \{\oslash\}$ where O is a set of all the possible operations. The function *alt* represents the means of merging two operations into a third with the property that the third is not in conflict with either of the first two. We say that two operations can not be merged if $alt(o_1, o_2) = \oslash$.

In order to define conflicts based on causality relations we can consider *state* to be some sort of vector clocks that summarize the knowledge of the user when attending the operation, while *alt* would be undefined. But in this definition we have the freedom to alternate *state* from user generated intent description to artificial intelligence algorithms, depending on how the system deals with conflicts and what sort of conflict tolerance it has. The complexity of the conflict definition reflects directly in the guarantees the system is able to make about consistency and other observable metrics, therefore most of the systems will probably stay on the safe side, but still prefer to relax causal dependency.

Since, unlike in the case of the DCSP, we are not trying to solve a set of constraint that may not all fit on an agent but rather satisfy the single constraint of consistency, we can use specific predicates defined on the data existing on each node and define conflicts based on the evaluation of these predicates. We consider *consistency* to be a predicate defined on $O \times O$ that is evaluated to *true* if the two operations can be applied in parallel and preserve the intent of both. We define the conflict between two operations and write $o_1 \otimes o_2$ if

$$\neg consistency(o_1, o_2) \wedge alt(o_1, o_2) = \oslash$$

.

As it can be seen from the definition, there are simplified implementation of the predicates and functions that will lead to all the conflict definitions that were described earlier. Also, the definition allows us to further define and formalize the conflict resolution algorithms.

4. Conflict resolution types

Once a conflict is detected, it must be resolved in order for the system as a whole to function properly. Most of the systems that allow conflicts to appear between parts have a certain built in tolerance for conflicts, but that usually degrades the quality of the service or renders it useless. Since the degradation is acceptable because the cost of avoiding it altogether is prohibitive, the system must ensure that the conflicts are resolved as quick as possible, or, at the very least, ensure that the conflicting situation will not exist forever in the condition of normal functioning of the system.

We identify the following characteristics of any conflict resolution algorithm that can be used in order to classify and better understand the consequences of using them:

- *Convergence* - The guarantee that the system will converge from a conflict state to a conflict free state in a bounded interval of time. It is very difficult to work with a system that does not guarantee convergence in at least some cases. While such systems where perpetual conflict situations are accepted could exist, usually there are guarantees like "system converges if it is partition free, all the components are up and no additional conflicts are added". It is very important for a conflict resolution algorithm to clearly state the conditions under which it will converge, since convergence under any conditions is impossible.

- *Performance* - One of the reasons to implement conflict tolerance in the first place is to decrease the cost of the system, therefore performance is an important characteristic. The performance level can be measured in many ways and the relevance of these metrics depend on the specific application. Possible metrics include the time, the network resources that are used, the number and type of resources that are involved (for example an algorithm that requires a human to make decisions will probably be more expensive than one that relies entirely on computers).
- *Impact on non-conflicting state* - Most of these systems accept conflicts because they are supposed to be rare, and an optimistic approach to resolving them is applicable. However, if the enforcement of the optimistic concurrency will affect all operations, including the majority that will be conflict free, the price might be prohibitive. One might argue that this is just another possible type of performance metric, but we would rather consider it as a separate category because performance is strictly bound to the process of resolving a conflict situation.
- *Intention preservation* - This characteristic takes into consideration the source of the conflict. Usually conflicts are not introduced by errors in the system but by users or systems working independently and having only a partial knowledge of the environment. In such cases, it is important to determine how and to what degree is the intent of each state that is in conflict being preserved by the conflict resolution algorithm. For example, if a document contains a circle, a user can alter the document to become a "happy face" while another could change the circle to a square. The result of having a square "happy face" will not preserve the intent of either user.

5. Conclusions and future work

In this paper we unified approaches from various communities involved in the research of distributed systems and created a generic definition of what a conflict is. Another contribution is the enumeration of several aspects that are important in the conflict resolution phase. As a system can not remain in a conflict state forever, all systems need to address the issue of conflict resolution, therefore, identifying the characteristics of a conflict resolution algorithm is of utmost importance.

We have used the conflict detection and resolution analysis from this paper in order to implement the JStabilizer ([4]) system and model the object oriented framework so that it accommodates a wide range of possible usage patterns. We will continue to develop the JStabilizer framework and implement more test cases that will refine and reinforce the validity and generality of the definitions and concepts included in this paper.

References

[1] Pauline M. Berry, Tomás Uribe, Neil Yorke-Smith, Cory Albright, Emma Bowring, Ken Conley, Kenneth Nitz, Jonathan P. Pearce, Bart Peintner, Shahin Saadati, and Milind Tambe. Conflict negotiation among personal calendar agents. *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems - AAMAS '06*, page 1467, 2006.

[2] E.A. Brewer. Towards robust distributed systems. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, 19:710, 2000.

[3] T.D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, page 407, 2007.

[4] Costa Ciprian. JStabilizer code repository (http://code.google.com/p/jstabilizer/), 2009.

[5] S. Citro, J. McGovern, and C. Ryan. Conflict management for real-time collaborative editing in mobile replicated architectures. *Proceedings of the thirtieth Australasian conference on Computer science-Volume 62*, page 124, 2007.

[6] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16:133–169, 1998.

[7] C Sun, X Jia, Y Zhang, Y Yang, and D Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer Human Interaction*, 5:63–108, 1998.

[8] Chengzheng Sun and David Chen. Consistency maintenance in real-time collaborative graphics editing systems. *Interactions*, 9:1–41, May 2002.

[9] D. Sun, S. Xia, C. Sun, and D. Chen. Operational transformation for collaborative word processing. *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, 6:446, 2004.

[10] Werner Vogels. Eventually consistent. *ACM Queue Communications*, 2008.

[11] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10:673685, 1998.

Babes-Bolyai University, Department of Computer Science, Cluj-Napoca, Romania

E-mail address: costa@cs.ubbcluj.ro