STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVI**, Number 1, 2011

TOWARDS A MIDDLE LAYER FOR LARGE DISTRIBUTED DATABASES

COSTA CIPRIAN

Abstract. This paper addresses the lack of a cheap reusable software component for building large, database backed, distributed systems. Even after these systems have become mainstream through their use in companies like Google and Amazon, there are still difficult to approach for a small group of programmers with limited resources. We intend to have the JStabilizer framework fill this empty spot and we present in this paper some of the problems we would like to simplify solved and a short description of the solution.

1. Introduction

In many cases large distributed databases and services in general prefer to use cheap hardware and deal with the failures in the software, as failures will occur regardless of how expensive the hardware is. The main problem is that, even if the cost of hardware has decreased, the cost of the software is much higher since it is non trivial to write software that operates within some correctness definition even in the presence of failures in the system. We see decreasing the software cost as the next step in making large distributed system a commodity and we started implementing JStabilizer([3]) with this purpose in mind.

In the first part of this paper we present 4 use cases that we consider representative for the problem at hand and analyse the impact of the derived requirements and the differences between the approaches programmers could have when implementing such a distributed system.

In the second part we present an high level overview of JStabilizer as it is implemented now, explain some of the basic flows within the system and give some examples of how we envision the usage of this middle layer framework.

Received by the editors: March 19, 2011.

2010 *Mathematics Subject Classification.* 68P15, 68M14.

1998 *CR Categories and Descriptors.* C.2.4 [**Distributed Systems**]: Subtopic – *Distributed databases*;

Key words and phrases. distributed systems, middle layer.

2. Use cases

The main problem we are trying to solve is the introduction of a framework that would allow regular programmers to work with distributed databases without any strong synchronization requirement. Giving up on synchronization and commiting all transactions locally has the advantage of making the system available even in the case of partial failure (for example, a web server taking order from clients will still be available even if it becomes isolated from the rest of the network). The big problem with this approach is that the distributed database might (and will) become inconsistent.

In this first part of the presentation we will discuss what we consider to be 4 representative use cases where it is preferable to accept the inconsistencies and design the software to be resilient to those inconsistencies as opposed to creating a system with a smaller availability guarantee.

The first use case discusses the use of a distributed database on multiple independent agents. The problem bares some resemblance to DCSP - Distributed Constraint Satisfaction Problem ([11], [10]) but differentiate itself as it studies this in the context of a database and instead of finding the right assignment to some variables in a negotiation process the goal is to achieve consistency of the database.

The second use cases discusses the implementation of a distributed message board. The difference from the first use case is that instead of having conflicts between the updated values of some entity, the conflict is between the order in which replies appear in the discussion threads.

The third use case presents the well known problem of airline ticket reservation. In these commercial systems there is a strong requirement for availability because of the large income that is lost if, for example, the customers would not be able to purchase a certain item for a certain period of time. The difference here is in the conflict resolution process which requires human intervention.

The final use case discusses mobile environments which are known for their high volatility. A lot of research was done in this area ([1], [4], [6], [7]) but they usually address only specific use cases without trying to unify concepts with those found in server side distributed systems.

2.1. **Agent database synchronization.** We considered and implemented the example of a restaurant where waiters are robots that are connected in arbitrary ways and without any single point of control. The example can be generalized to any type of independent agents that operate based on information from a database that is supposed to be synchronized.

In this case the database would contain the assignments of waiters to tables, so each waiter will know who is assign to which table. When the client

enters the restaurant and sits down at one table, some of the waiters will want to assign the table to themselves, thus introducing an inconsistency in the database (the consistency constraint is that only one waiter should be assigned to one table).

This is very similar to the self stabilization problem that was introduced by Disjkstra ([5]). The difference is that in this case there is no token that is passed from node to node in order to make sure that only one of them "moves".

It is important for each waiter to commit the transaction locally because it has to start working towards the goal of serving the table without requesting confirmation from any other peer (as this confirmation may never arrive because of some system failure and the table would remain without a waiter). It is obvious that the database may not become consistent as long as there are failures in the system, for example, two waiters that are completely cut off from the rest of the system might independently decide to attend a table and the system will not be able to correct this situation until the waiters come back online). In this case the programmer will just have to accept the fact those conflicts can appear and deal with them - the waiter will have to understand the fact that the table was already taken by another peer (ignoring this state would probably end up with a waiter constantly asking for the order from the client and constantly being turned down). As seen from this example a characteristic of these systems is that the cost of fixing an inconsistency is much smaller that the cost of making a system that is always consistent (in this case lots of redundant hardware and failover policies) or accepting temporary unavailability in the system (in this case nobody attending a table due, for example, to the inability of the system to form a majority because of partitioning or a failure in the control process).

In such a distributed system we would like the local database to always be consistent (the consistency predicate always holds on the local database) - in our case, each waiter will always have a database that always assigns a single waiter to a table. Not being able to rely on this constraint being true would make programming the system extremely complicated and costly. In other words we can deal with waiters that are wrong, but we can not deal with waiters that are inconsistent. The way to have this property and still be able to work towards the goal of having the database globally consistent is to create inverse operations for each operation that we would normally have. In this example the inverse operation would be deciding to no longer attend a table due to some sort of information that is received from peers. The key problem here is making sure that the system is doing progress - we do not want waiters to all decide that they are going to attend a table and then all

apply the reverse operation and decide not to attend. We will see later that ensuring progress is being made is not a trivial task.

A informal presentation of the algorithm we came up with is:

- Once a table is occupied several robots will evaluate a probabilistic function and decide if they should attend the table. Once the decision to attend is made, the robot will start working towards its implementation - actually attending the table.
- Each waiter that decided to serve a table will inform his peers of the fact that it is attending the table and also include the number of tables that are assign to him (according to its local database.)
- If a waiter receives the above message one of two things can happen:
  - No conflicts are detected - the information is saved to the local database and the message is forwarded to other peers.
  - A conflict is detected - in which case the conflict has to be resolved prior to saving the information to the local database and forwarding the message. The conflict resolution algorithm is the following:
    * The waiter with the smaller number of total tables wins, if the numbers are different.
    * The youngest waiter wins if the total number of tables are the same (we assume that no two waiters were born at exactly the same time)

Some observations on the proposed algorithm:

- A key assumption is that the system is asymmetric. Without this property a waiter would not be able to choose a winner of a conflict and assume that others will choose the same winner if presented with the same conflict
- The number of tables served by the waiters that are in conflict is not computed based on the information available in the local database, it is taken from the messages received from each of the waiters. As asymmetry, this is a requirement for the convergence of all the independent conflict resolutions performed by each peer.
- The scope of the algorithm is not to come up with a fair distribution of tables to waiters. The "fairness" is only used as a criteria for conflict resolution - could be replaced with any other criteria as long as each waiter resolves the same conflict in the same way.
- The algorithm is easily provable as correct if, for example, there are only two waiters that decide to serve at a table and the waiters are arranged in a circle. However, things get more complicated when the structure or the number of waiters is varied - each intermediary waiter

could resolve a different subset of the total set of conflicts in the system and, as such, introduce new conflicts.
- Conflict detection is an important part of the algorithm. Based on what is defined as a conflict, the algorithm might become stable (always converging to a distributed consistent state), or unstable (capable of remaining in an inconsistent state even without the presence of failures in the system).

2.2. **Distributed message board.** Another example that we considered when analysing the possible approaches to this problem is a distribute message board. The concept is similar to the one used in the USENET service - the message boards are stored in multiple places, comments are made on each server individually and then replicated from one server to another.

We can consider this a problem of ensuring the consistency of a virtual database that contains all the messages in all the servers in the network. The difference from the waiters problem is that in this case the conflicts are given by the order in which the messages appear, not by the content of the messages themselves. From a user perspective the requirement is that once a user has seen a specific order of messages, that order will not change, regardless of what other servers are doing. An example of a possible conflict is the following:

- The database starts with message A.
- On server S_1 a user posts a reply B.
- Before receiving the reply B from S_1, a user from S_2 will post another reply C to A.
- Servers S_1 and S_2 start propagating the actions and server S_k and S_j detect the conflicts at the same time.

Any of the sequences $A \to B \to C$ and $A \to C \to B$ are correct, but both S_k and S_j need to choose the same order, otherwise the servers will enter in an inconsistency that can not be solved while satisfying the user constraint of never seeing messages with different ordering.

From an implementation point of view we see the following differences when comparing this with the waiters example:

- Not all conflicts can be resolved right away, some might have to wait for other data to arrive at the node. For example S_1 creates message A, server S_2 creates B as a reply to A and server S_3 creates C as a reply to B. It is possible that server S_4 receives the message containing the creation of C before it receive the message for B and, as a consequence has to wait for B to arrive.
- There is a granularity difference between the objects on which we evaluate consistency (the entire thread) and the information that is exchanged between servers (the messages composing the actual thread).

While it is possible to always exchange the entire thread, there is no valid reason to do it from a conflict detection and resolution point of view.

2.3. **Airline ticket reservation.** Another use case that presents some different aspects of the problem is an airline ticket reservation system. Consider a database with all the flights in a specific area and distribute that database on several servers. Each time a client reserves a seat, the transaction commits locally, so there is a possibility that a seat is reserved twice - which is clearly an inconsistency. Solutions for reducing the number of conflicts can be found, like, for example, each server could provision some number of seats and, when interested on reserving more, it could ask from other servers to reduce their provision but, if the database is big enough and the infrastructure has a high enough probability for failure, we will end up with empty seats that can not be reserved.

The only solution is to allow distributed conflicts to exist (while maintaining each database locally consistent). Once a conflict is detected it is sent to a human operator which then negotiates with the involved parties in order to come up with a mutually accepted solution. An actual implementation was discussed by S.Bourne and Bruce Lindsay ([2]) in the context of the Sabre system where a tolerance of approximately one duplicate for every 1000 reservation was accepted and it proved to be cheaper to resolve those conflicts (by promoting some of the conflicts to first class or paying for an extra day of vacation for the customer) than to make sure that the system is always in a consistent state (which can only happen if the system has a lot of redundancy and state of the art equipment or if complete failures of the system are acceptable).

When comparing with the other solutions the following aspects stand out:

- The messages that are exchanged between servers do not contain enough information for the conflict to be resolved locally by each server, so some notion of special authority that is capable of resolving the conflicts is required.
- Even if a special authority exists and is a single point of failure (if it fails the database will never become consistent), it does not affect the reservation service - the clients are still able to reserve seats.
- There is a time constraint on the conflict resolution process. While in the previous cases it was accepted for the conflict to be detected only when the waiter reaches the client or there was no real penalty for posts to not propagate to all servers, in this case, the system is not going to be used if the probability for the conflicts to only be detected when the clients reach their seats is not statistically negligible. But

the probability of conflicts and upper limits on the expected levels of failures can be imposed and can be lowered through traditional techniques - the important thing is that the reservation system is always available to the client and no one goes to the competition because the web server throws an error message when a reservation is made.

From an middle layer implementation point of view the following aspects are important:

- Unlike the distributed message board, where some of the conflicts were not solvable when detected, in this case the resolution can not be made locally at all. Once a conflict is detected is has to be forwarded to a special authority which will resend a message with the resolution before that takes precedence in front of any other message from any other system.
- Depending on how the information is transmitted, it is possible for:
  - A conflict resolution to be received by a server before it detected the conflict
  - A conflict to be reported after it is resolved. This can happen if the conflict was previously detected by another server, resolved by the authority, but the current server found the conflict before receiving the resolution.
- If an incoming message resulted in a conflict then it makes no sense to forward the message as a message with the reservation that is in the local database was sent earlier and the current message would just be detected as a conflict by all servers we could send it to. This does not mean that a conflict can not be detected by several servers at once.

2.4. **Contact synchronization.** A final use case that adds something new to our system is an application for contact synchronization on mobile devices. We could consider a database with all the contacts on all the phones and have an application that creates relations between contacts from different phones. In such a setup an inconsistency could be defined as a difference between the values of two related contacts on different phones.

Following the algorithms described above, in this case, when someone updates a contact the change will propagate to connected phones. Given the nature of this application, the programmer could choose for example to allow the user to accept or deny an update, or the choice could be to always accept the change if there is no conflict. Another important aspect that the programmer might want to control is how the updates are forwarded. Should an update be forwarded even if it was not accepted? A reason why it could be forwarded is to allow all the network to receive the update regardless of the distance between nodes and then propagate the accept/deny option chosen by

the user in order to present other users the number of people in the network that accepted or denied the update.

Another aspect that is different is the conflict detection and resolution mechanism. The conflict detection could be more complicated than a simple value comparison. For example two people might update the phone number of the contact to the same number, but one will use the country code while the other would not, or there could be other syntactic differences that would not qualify as conflicts. Following this argument, the programmer might want to use a simple value comparison and then ask the user if any differences are actually conflicting. Conflict resolution could be done on each phone by each user or the application could forward the messages and let the users who generated the conflict agree on a value.

When comparing with the other examples some differences stand out:

- The conflict resolution strategy does not guarantee that the system stabilizes if failures are fixed an no other conflicts are generated. A simple example would be the classical live lock scenario: two users generate a conflict and, when each user detects the conflict, it can resolve the conflict by choosing the other version. This way the number of conflicts is not decreased and the database remains inconsistent.
- Failures could be permanent. In previous scenarios, the system were more or less under the control of the programmer or of parties that are directly interested in the correct functioning of the system as a whole. In this case the control is entirely in the users hand and some one could just loose their phone or uninstall the application
- Conflict resolution might require a special authority but, unlike the airline ticket reservation system, in this case it is not an external service but instead is a system formed from the users who generated the conflict in the first place. This behaviour introduces the following particularities in the conflict resolution mechanism:
  - It is possible that the special authority in a conflict resolution to have permanent failures. This is different from the airline ticket reservation scenario where we assumed that the conflict resolution authority only suffers transient failures.
  - If more than two users enter a conflicting state, the system should not require all of them to agree on a conflict resolution because this would allow a permanent failure in one of the parties to prevent other users from doing progress. Furthermore, the system should not require the users to actually agree on one version, it should just present each of the user with a list of conflicts they generated and ask them to choose a resolution.

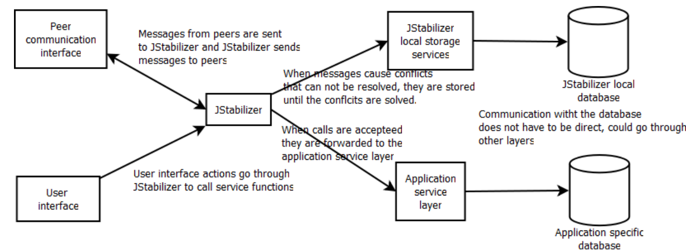

FIGURE 1. Overall system presentation.

While the value of such an application could be argued and is subject to personal preferences, the scenario and the requirements could come up in this or other applications of this sort and the needs could be addressed by the same middle layer used in the other scenarios.

2.5. **Wrap up.** From these examples we can extract some of the requirements for a middle layer that would allow them to be implemented at a smaller cost:

- Communication - while the actual means of communication are irrelevant and could be anything from message queues to shared memory systems, the programmer should be able to decide when the updates are propagated to peers.
- Conflict detection - some applications do not require a high degree of sophistication in this area but others might choose to implement more subtle strategies where a simple last modification timestamp or a versioning scheme might not be enough.
- Conflict resolution - we identified some of the conflict resolution scenarios (local, global special authority, per conflict special authority) but there could also be others, so this is a likely extension point. The middle layer must also be prepared from conflict resolution strategies that are not guaranteed to make progress towards the global consistency goal - there are systems that are still meaningful if such a goal is never reached.
- Special storage - some systems will not be able to apply the incoming messages on the local database immediately, so the middleware needs to have some sort of storage mechanism to ensure that these messages will be retrieved when required. The reason why these updates will have to wait is that the local database is always consistent, so, as long as a conflict is not resolved, the changes can not be applied.

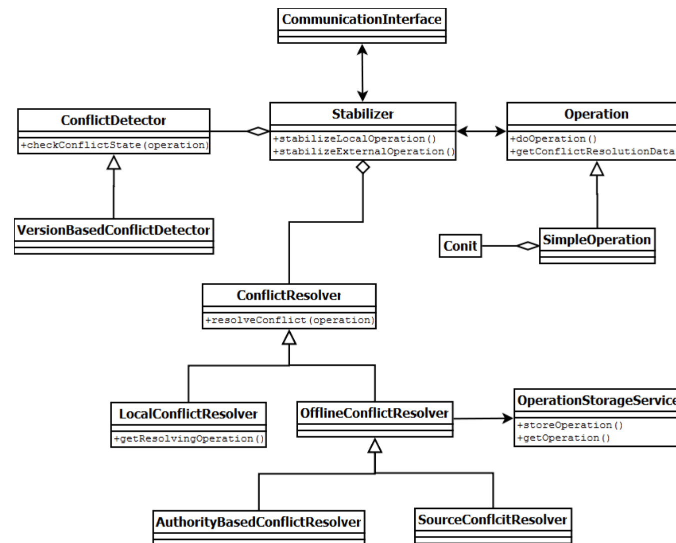

FIGURE 2. JStabilizer internal structure.

3. IMPLEMENTATION DETAILS

We started implementing JStabilizer as a middle layer framework that would lower the complexity of implementing the above described applications. The basic idea is to place JStabilizer in front of the service layer of the application, so all calls to the service layer will be registered with JStabilizer and, furthermore, JStabilizer will also receive information from the neighbour peers and send it to the service layer. We do not intend to make distribution transparent to the programmer for various reasons - a good presentation on how being transparent is forcing programmers to pay a high price and basically give up on the advantages of concurrency can be found in [9] (as a testimony to the controversy surrounding this issue, one of the authors went on to write such a transparent framework and explain the special circumstances when this is a good solution in [8]).

An overall presentation of how JStabilizer fits in the application is provided in figure 1. The programmer will have to make sure that all operations that work on items that are distributed will go through JStabilizer and program the services layer so that it supports update calls from both the direct users of the server and other servers in the network. Also the programmer will have to provide an implementation for the offline storage services required by JStabilizer (default implementation will be provided).

In figure 2 we have a brief presentation of the internal structure of JStabilizer. The main class is *Stabilizer* which is the gateway for all the operations that are executed and manages the temporary storage and the conflict resolution process. Each operation knows how to execute itself (call the proper service functions) but is passed to the stabilizer which will decide when and if it is actually executed.

The stabilizer will use on of the *ConflictDetector* and *ConflictResolver* classes in order to asses the state of each operation and proceed based on that.

A key concept in the implementation is the conit (CONsistency unIT) (introduced in [12] in a slightly different context) - defined as the smallest piece of information for which the study of consistency is significant. We use the concept of partial consistency - applications are more interested if a certain conit is consistent than if the entire database is consistent (for example the message board application can function correctly for threads that are consistent even if other threads in the database are not).

4. Conclusions and future work

We managed to present 4 representative use cases for the problem at hand that were used in testing the validity of the implementation and introduce the proper extension points. Most of the use cases were treated to some extend in other works but we present a unified solution for them. Also we inserted a brief presentation of the current JStabilizer implementation to get a sense of how the use cases have influenced it.

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] S. Bourne. A conversation with Bruce Lindsay. *Queue*, (November), 2004.

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

[4] 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.

[5] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[6] 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(1):63–108, 1998.

[7] 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(3):446, 2004.
[8] J. Waldo. Scaling in games \& virtual worlds. *Queue*, 6(7):10–16, 2008.
[9] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. *Lecture Notes in Computer Science*, pages 49–64, 1997.
[10] 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(5):673–685, 1998.
[11] Makoto Yokoo and Katsutoshi Hirayama. Algorithms for Distributed Constraint Satisfaction:A Review. *Autonomous Agents and Multi-Agent Systems*, 3(2), 2000.
[12] H Yu and A Vahdat. Design and evaluation of a continuous consistency model for replicated services. *of the Fourth Symposium on Operating Systems Design*, 2000.

Babes-Bolyai University, Department of Computer Science, Cluj-Napoca, Romania
E-mail address: costa@cs.ubbcluj.ro