Formal Mathematical Requirements (work in progress)

Tom Benedictus and Amadeo Ascó
Trifork Leeds
tom@trifork.com and aas@trifork.com

28th July 2014

Contents

List of Figures

List of Tables

Abstract

Building reliable distributed systems, given the CAP theorem where partitioning must always be considered, relays in the trade-offs between consistency and availability. Conflict-free Replicated Data Types (CRDTs) define replicated data types with mathematical properties that ensure absence of conflict and confer them Strong Eventual Consistency (SEC), a form of Eventual Consistency (EC) which it is a technique of compromise. Consistency is a property of the data, not the datastore, given that the rules to decide how to synchronise are business decisions. CRDTs have been created to ensure that we have a computer model for handling data which accommodates data's nature in a world of vast communication and no definite central place of storage. Using an Relational Database Management System (RDBMS) for these type of systems can result in systems where a large amount of the processing is required to circumnavigate the shortcomings of a model that is not fit for purpose. A significant project for mediating this dilemma is SyncFree.

This article provides a brief presentation of the natural language requirements, which shows the special nature of the paradigm shift forcing large-scale distributed applications away from RDBMS.

1 Introduction

EC is one key element in the concept for a contemporary replicated database, as presented in [8], [7], which takes into account the CAP theorem; theorem that is presented in [3] and studied further in [1, 5]. In fact, a distributed database comprises data traversing the cloud together with data residing on devices, not yet if ever, going to move to another storage which is here referred to as the edge of the network. Atomicity, Consistency, Isolation, Durability (ACID) are the classic requirements for a database. We want to retain these although we realise that they may be relaxed in some areas and they should be avoided in other areas. Atomicity indicates that either the entire transaction can be carried out or it will be rolled back to the state before it was even started. In a large-scale distributed application this criteria cannot be accommodated in a reliable fashion as it results in deadlocks. Consistency cannot be guaranteed as part of the atomic transaction. We are trusting EC, which has been a prolific topic of research [9], [13], [7], [2], and provisioning mechanisms for detecting persistent inconsistencies. EC is a weaker data consistency which will require a complex background algorithm for reconciling conflicting updates [11]. CRDTs have been designed to solve the need for reconciliation by using SEC [10], without the need of complex conflict resolution, roll-backs, or consensus. Also composites of CRDTs present the same properties, [4], [9].

2 Use Cases

Three use cases has been identified by Rovio representing significant issues and difficulties of extreme-scale sharing and consistency within online and mobile large scale entertainment application. Additionally, we have three distinct use cases from Trifork each representing their own application domain and each posing issues that require CRDTs.

We are here providing an overview to illustrate the nature of the requirements we are faced with in large-scale distributed applications without global synchronisation.

The general constants and variables that applied to all the following use cases are presented in Table 1.

Name	Description	Type
DC	It is the set of all DCs. d identifies one of the DCs, $d \in DC$, where $\|DC\|$ is the number of DCs.	$\mathbb{Z}_+$
DV	It is the set of devices. dv represents one of those devices, $dv \in DV$, where $\|DV\|$ is the number of devices.	$\mathbb{Z}_+$
$Nodes$	It is the set of nodes. n represents one of those nodes, $n \in Node$, where $\|Nodes\|$ is the number of nodes in a DC.	$\mathbb{Z}_+$
$Clients$	it is the set of clients and c represents a client in the system, $c \in Clients$, where $\|Clients\|$ is the total number of clients.	$\mathbb{Z}_+$

Table 1: General definitions.

3 Entertainment Applications

Rovio, being one of the leaders in online and mobile entertainment, has provided three unique use cases from their current applications. These use cases are also relevant for many other possible application domains and are as such highly relevant for establishing the requirements for CRDTs.

3.1 Advertisement Counter

Advertising platforms need to accurately record impressions and clicks, in order to analyse advertising data. They use distributed counters, which are challenging to implement in a dynamic, fault-prone environment. CRDT counters are a promising solution; the challenge is to scale to an extreme numbers of users.

Rovio's Ads service keeps track of impressions and clicks for ads per campaign/ad/country. Typically these counts have some upper bounds after which the ad should not be shown any more. The upper bound may consist of the sum of several counters (e.g show the same ad 50,000 times in the US, 10,000 times in the UK and 100,000 times in total), so it is not really feasible to enforce the upper bound on the data storage layer.

The main use for the tracking data is to control the rate of ads shown to the users. The campaign capacity should be spread evenly over the duration of the campaign instead of showing all the impressions during the first hour of the campaign. Therefore, the data must be updated in real time although a good estimate is normally enough.

3.1.1 Current Implementation

The system maybe represented by the various constants and variables that follow, Table 2.

Name	Description	Type
AD	It is the set of all advertisement-campaigns. a identify one of the ads, $a \in AD$, where $\|AD\|$ is the number of ads.	$\mathbb{Z}_+$

Name/Description	Type
$T_a, T_a^{start}, T_a^{end}$	$\mathbb{R}_+$
T_a is the duration of the campaign for ad a. T_a^{start} represents the beginning of the campaign and T_a^{end} the end, with $T_a = T_a^{start} - T_a^{end}$.	
$maxTotalViews(a)$	$\mathbb{Z}_+$
It is the maximum total number of times the ad a should be shown.	
$maxTotalViewsPerDC(a, d)$	$\mathbb{Z}_+$
It is the total number of times the ad a should be shown by DC d.	
$maxViewsPerDevice(a)$	$\mathbb{Z}_+$
It represents the maximum number of times the ad a should be presented on a device.	
$viewsPerDevice(a, dv)$	$\mathbb{Z}_+$
It is the number of times the ad a has been shown on the device g. $h(t)_{ag}$ is the same than h_{ag}.	
$verifiedViews(a, n, q)$	$\mathbb{Z}_+$
It is the verified number of times an ad a has been shown by node q as the node n report it, $n, q \in \{1, \ldots, \|Nodes\|\}$.	
$averageViews(a)$	$\mathbb{Z}_+$
It is the average number of times ad a is shown. The workload is equally spread between all the nodes, $\frac{averageViews(a)}{\|Nodes\|}$.	

Table 2: Ad Counters Constants and Variables.

Equality 1 states that the total maximum number of times an ad should be presented in each DC is equal to the maximum total number of times that ad should be shown in the campaign, and Inequality 2 states that the ad a must be shown on any device only a maximum number of times of $maxTotalViews(a)$. The total number of times that ad a has been shown on completion of the campaign is expressed by Equation 3. The goal of the system is to minimise Δ_a, $\Delta_a \in \mathbb{N}_0$.

$$maxTotalViews(a) = \sum_{d \in DC} maxTotalViewsPerDC(a, d) \tag{1}$$

$$maxViewsPerDevice(a) \geq viewsPerDevice(a, dv) \tag{2}$$

$$maxTotalViews(a) = \sum_{dv \in DV} viewsPerDevice(a, dv) + \Delta_a \forall\ t \geq T_a^{end} \tag{3}$$

The Ads service runs on multiple service nodes, so in order to avoid write conflicts each of those nodes has its own document for the impression and click counters in Riak. This is already a simplified implementation of the counter CRDT. The true value of the counter can be obtained by calculating the sum of all counters manually.

Even if the current solution works, it is neither very elegant nor easy to maintain. The CRDT counters will provide a more efficient solution for updating the counters. The ad system does not need a strict limit for the counter and we can implement an optimistic solution: if the counter value is less than the limit then show the ad and increase the counter. This may result in showing the ad too many times, but it is "close enough". Client applications running on different kinds of devices (mobile phones, tablets, etc.) connect to the Ads service in order to determine which ads to show to their users. The same ad should not be shown on the same device any more than 3 times a day ($maxViewsPerDevice(a, dv) = 3$), so the Ads service keeps track of on which device has been shown which ad and when. This data is currently stored in one document that contains information on the ads shown on a device during the last couple of days (basically which ad has been shown and when). It might be possible to utilise CRDTs in this document.

In a DC each server node has its own cache representing the counters. The cache could be dropped theoretically, but in practice it will be needed. The counters get synced between the life statistics nodes via Riak. Each counter will have some copies, e.g. three, on the database meaning the value is formed based on those copies. The average number of times, ad a is shown, is $averageViews(a)$ for an interval of time, so the estimated number of times the ad a has been shown is represented by Equation 4 as seen by node n, note that only one DC is currently used. Also the verified number of times an ad has been shown, $verifiedViews(a, n)$, is the number of times node n reported to have shown the ad a in the last synchronisation at previous period of time.

$$views(a, n) = \sum_{q \in Nodes} verifiedViews(a, n, q) + averageViews(a) \tag{4}$$

The sum of all the times an ad a has been seen corresponds to the real number of times the ad a has been shown by the system, as shown by Equation 5. The total number os times the ad a has been seen at any time, $views(a, n)$, as seen by a node n may be less or equal to the real total number of times the ad a has been shown, $views(a)$, as shown by Inequality 6. This inequality should be eventually (after last synchronisation) an equality.

$$views(a) = \sum_{n \in Nodes} verifiedViews(a, n, n) \tag{5}$$

$$views(a) \geq views(a, n) \tag{6}$$

Also the numbers of times an ad a has been shown after the ad-campaign has concluded must be the same irrespective of the node reporting it, as shown by Equation 7.

$$views(a) = views(a, n) = views(a, m) \forall\ n, m \in \{1, \ldots, |Nodes|\}, t \geq T_a^{end} + \Delta t \tag{7}$$

The life statistics nodes serve the entire state of all campaigns. They hold a cache for the data and keep it in sync via Riak. There is a local estimation in place between the nodes synchronisation since the data is not synced on every update they get from the ad servers.

3.1.2 Multiple Data Centres

The campaign counter could also be split between the different DCs in each country which will improve the accuracy of the counter, Figure 1. At each DC a part of the overall counter would be assigned to each DC, which will only be able to service this number of times the specific ad. The changes to the ad counters are replicated in the other DCs at different intervals.

The distribution of the counter could be based on different criterion, e.g. population size covered by each DC, existing statistics from previous campaigns or studies, costumer preferences, intention of the

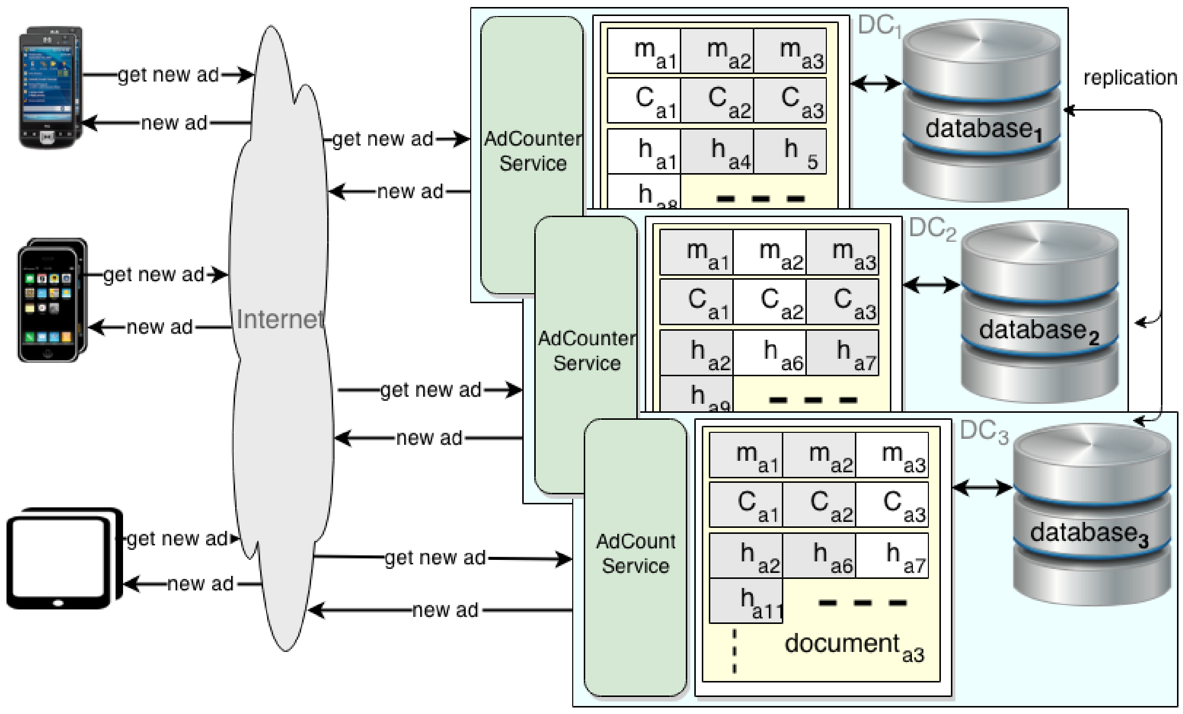

Figure 1: Overview for the distribution of counters with three DCs, $|DC| = 3$, $m_{ad} \equiv targetMaxViews(a, d)$, $C_{ad} \equiv maxViewsPerDevice(a)$ and $h(a, g) \equiv views(a, d, g)$.

campaign like the increase in the market share in an already established part of the territory or entering a new area to extend the territory covered.

It may be decided that when a device serviced by a DC changes to be serviced by another DC, e.g. by changing location, then its counter is now know by the new servicing DC, potentially showing those ads already seen again, otherwise the device counter needs to be replicated between DCs, but not necessarily to all DCs.

The extra constants required for multiple DCs are presented below, Table 3.

The model needs to be extended by the following expressions:

$$targetMaxViews(a, d) \geq views(a, d) \tag{8}$$

$$views(a, d) = 0 \;\forall\; t \leq T_a^{start} \tag{9}$$

$$targetMaxViews(a, d) = views(a, d) \;\forall\; t \geq T_a^{end} \tag{10}$$

$$totalMaxView(a) = \sum_{d \in DC} targetMaxViews(a, d) \tag{11}$$

$$totalViews(a) = \sum_{d \in DC} views(a, d) \tag{12}$$

Inequality 8 states that the counter $views(a, d)$ always has a limit which it is the maximum number of times the ad d must be shown by DC d, whereas at the beginning of the campaign no ads has been shown as stated by Equation 9, and once the campaign has finish the total number of times an ad has been shown by each DC is the same than the maximum allowed as expressed by Equation 10. Equation 11 states that the ads are distributed through out all the DCs. Equation 12 states that the overall total number of times an ad a has been shown is equal to the sum of the number of times the ad a has been shown by each DC.

$$maxTotalViews(a) \geq \sum_{d \in DC} totalViews(a, d) \tag{13}$$

Name/Description	Type
$Nodes_d$	$\mathbb{Z}_+$
It is the set of nodes in DC d. n represents one of those nodes, $n \in \{1, \ldots, \|Nodes_d\|\}$.	
$averageViews(a, d)$	$\mathbb{Z}_+$
It is the average number of times ad a is shown for a given time interval by DC d. The workload is equally spread between all the nodes in the same DC, $\frac{averageViews(a,d)}{\|Nodes_d\|}$.	
$verifiedViews(a, d, n)$	$\mathbb{Z}_+$
It is the verified number of times an ad a has been shown by node n in DC d as the node n reported to have shown the ad a in the last synchronisation at previous period of time.	
$targerMaxViews(a, d)$	$\mathbb{Z}_+$
It is the maximum number of times the ad a should be shown by DC d. This may as well be used to restrict the locations (represented by L), e.g. country an ad is shown. DCs outside these locations will not show the ad so $targerMaxViews(a, d) = 0 \ \forall \ a \notin L$. Also the replication will only be necessary between the DCs with $targerMaxViews(a, d) > 0$.	
$views(a, d)$	$\mathbb{N}_0$
It is the total number of times the ad a has been shown on devices by DC d from the beginning of the campaign, T_a^{start}, to time t.	
$totalViews(a)$	$\mathbb{Z}_+$
It is the overall total number of times the ad a has been shown from the beginning of the campaign, T_a^{start}, to time t.	

Table 3: Ad Counters extra Constants and Variable.

$$maxTotalViews(a) = totalViews(a) + \Delta_a \ \forall \ t \geq T_a^{end} \tag{14}$$

Inequality 13 gives the total limit for all the $totalViews(a, d)$ which can be obtained from Inequality 8 and Equation 11, whereas Equation 14 shows that the total number of times ad a has been shown, once the campaign has concluded $t \geq T$, is equal to the total number ad a should has been shown.

Also Equation 4 can be generalised for many DCs as shown in Equation 15.

$$views(a, d) = \sum_{n \in Nodes_d} verifiedViews(a, d, n) + averageViews(a, d) \tag{15}$$

There is the possibility that a device in the border between two DCs, where the ad is run, receives more than its limit if the two DCs are out of sync for that device.

To have into account the state of the data in each of the DCs the previous definitions are extended and some new introduced below, Table 4. The discrepancy of the counter for ad a shown on devices by

Name/Description	Type
$views(a, d, q)$	$\mathbb{Z}_+$
It is the total number of times the ad a has been shown on devices by DC d from the beginning of the campaign to time t as it is seen by DC q, $d, q \in \{1, \ldots, \|DC\|\}$.	
$\Delta views(a, d, q)$	$\mathbb{Z}_+$
It is the discrepancy of the total number of times the ad a has been shown on devices by DC d from the beginning of the campaign to time t as reported by DC q, as it is represented in Equation 16.	
$\Delta totalViewsDiscrepancy(a, d, q)$	$\mathbb{Z}_+$
It is the absolute total discrepancy of the overall total number of times the ad a has been shown from the beginning of the campaign to time t when using the values provided by DC d, which it is represented in Equation 18.	

Table 4: Ad Counters Constants and Variable for discrepancies in values between DCs.

DC d is zero when it is reported by the same DC as expressed in Equation 17.

$$\Delta views(a, d, q) = views(a, d, d) - views(a, d, q) \tag{16}$$

$$\Delta views(a, d, d) = 0 \tag{17}$$

$$\Delta totalViewsDiscrepancy(a, d) = \sum_{q \in DC} \mid \Delta views(a, d, q) \mid \tag{18}$$

The total counter is said to be consistent throughout all the DCs if there is not any discrepancy between all the DCs, as expressed in Equation 15.

$$\Delta totalViewsDiscrepancy(a, d) = 0 \tag{19}$$

3.1.3 Notes

The number of times an ad should be shown on a device (formally $numViewsPerDevice(a)$) could also depends on the DC associated with it, $numViewsPerDevice(a, d)$, as shown in Figure 1.

Another consideration is what happen when a device moves between DCs. An approach would be that when the device moves from one DC to another its details are also migrated to the new DC, of course if the connection is available. Otherwise $h(a, d, g)$ would be $h(a, g)$.

Some options:

- The new associated DC will create a record for the device without knowledge of the device previous association to any other DC (simple case).
- The details of the device are moved from the old to the new DC for when there is connection between DCs, otherwise apply previous option. This could also be approached in multiple ways, some of which are:
  - The device stores on its own internal storage the DC it is associated with and passes the DC servicing it within each transaction so if it is serviced by a different DC that DC will try to get the device's data from the current associated DC. Also any response will return the ID of the DC that replied which will be used in the subsequent request submitted by the device.
  - Each new association provides new clean details so a device that moves from one DC to another could potentially see the same ad twice the maximum allowed number of times per device, Inequality 20.

$$h(a, d_1, g) + h(a, d_2, g) \leq maxViewsPerDevice(a, d_1) + maxViewsPerDevice(a, d_2) \tag{20}$$

3.2 Leader Board

Leaderboards are used in games to provide information on who are the best players globally (and often also locally) and how the current player ranks against other players. Rovio's leaderboard service provides a different kind of leaderboards for games. The default type of leaderboard is level-based which means that the high scores are stored by level, so each user has one document per each level passed in a game. Each level score document contains the user's highest score, (estimated) rank, matchmaking and percentile indices and some other additional properties the service itself doesn't care about (e.g. stars, lap time etc. depending on the game). Since the leaderboard should always contain the highest score the user has achieved in a level, custom conflict resolution based on the high score is required. With CRDTs the conflict resolution could be done so that the maximum or minimum score wins, and the rest of the properties are taken from the update that contained the new score (no conflict resolution required). The leaderboard service supports the following operations:

1. Send score (adds or updates the high score of the user)
2. Get ranking (returns the user's ranking in a level)
3. Get matching (returns a list of user IDs whose ranking is close to the requesting user)
4. Get leaderboard (returns the leaderboard for top ranking users, user's friends etc.)

Name	Description	Type
$Games$	It is the set of games and g represents a game in the overall system, $g \in Games$, where $\|Games\|$ is the number of games.	$\mathbb{Z}_+$
$Levels_g$	It is the set of levels in game g and l identify a level in game g, $l \in Levels_g$, where $\|levels_g\|$ is the number of levels in game g.	$\mathbb{Z}_+$
$Players_{gld}$	It is the set of players which have played at some stage the game g at level l, as it is seen by DC d, and p identifies one of those players, $p \in Players_{gld}$. Where $\|Players_{gld}\|$ is the number of players which have played at some stage the game g at level l.	$\mathbb{Z}_+$
$Score_{gldp}$	It is the score for player p has achieved at level l of game p as seen by DC d.	$\mathbb{Z}_+$
$highestScore_{gld}$	It represents the highest score achieved by all players that have played game g at level l as it is seen by DC d, which calculation is shown in Equation 21.	$\mathbb{Z}_+$
$Players_{gld}$	It represents the group of all the players which scores for the game k at level l are not lower than the scores achieved by any of the other players which have played the game g at level l as it is seen by DC d, represented in Equation 22.	$\mathbb{Z}_+^n$

Table 5: Leader Board Constants and Variables.

The game clients usually use the same DC for every game session so global consistency could be lowered and higher consistency required within the same DC. This would mean the global leaderboard would be updated with a longer delay than the country-specific one but that shouldn't really matter.

A mathematical representation of this use case is represented below, where Table 5.

$$highestScore_{gld} = \max\{Score_{gld1}, \ldots, Score_{gld|Players_{gld}|}\} \; \forall g \in Games, l \in Levels_g, d \in DC\} \tag{21}$$

$$Players_{gld} = \{j | Score_{gldj} \geq Score_{gldq} \; \forall q \in Players_{gld}\}\} \tag{22}$$

Furthermore $Players_{gld}$ could be extended to represent the different positions in the Leader Board, such that $Players^i_{gld}$ is the group of players which are at position i on the Leader Board, $i \in \mathbb{Z}_{\geq 1}$, as shown in Equation 23. This means that $Players^1_{gld}$, which represent the top position, is equivalent to $Players_{gld}$, such that $Players^1_{gld} = Players_{gld}$.

$$Players^i_{gld} = \{j | Score_{gldq} < Score_{gldj} < Score_{gldo} \; \forall q \in Players^{g-1}_{gld}, o \in Players^{g+1}_{gld}\}, g \in \mathbb{N}_{>1} \tag{23}$$

Equation 24 expresses that for any player within J_{kli} their highest scores for game k at level l, as it is seen by DC i, is equal to the highest score achieved between all the players for that game an level as it is seen by DC i.

$$highestScore_{gld} = Score_{gldp} \; \forall \; g \in Games, l \in Levels_g, d \in DC, p \in Players^1_{gld} \tag{24}$$

3.3 Virtual Wallet

Virtual Wallet applications manage virtual economies.The clients keep a local state and also does credits and debits to their local state but the clients local state cannot be trusted. Such applications require massive scalability and very robust security guarantees. To ensure very low per-transaction financial cost, as required for use at a ne granularity (nano-transactions), some consistency constraints may need to be temporarily relaxed, but not others. The challenge is to maintain correctness at an extreme scale, i.e., ensure money does not vanish nor is created out of thin air, despite data fragmented across numerous replicas, lost or duplicated information, long-term disconnection, etc. Rovio's Wallet service provides a delivery mechanism for in-game items and manages the user's virtual currencies. A single wallet contains balances of the virtual currencies the user owns, vouchers for the in-game items (e.g. powerups) the user has purchased but have not been delivered yet, and a transaction log that lists the most recent transactions for the wallet.

- Balance consists of a numerical value and currency name (e.g Crystals: 150 or Euro: 2.45).
- Voucher consists of a unique voucher ID and item details (item ID, name etc.). Vouchers are removed from the wallet when consumed.

- Transaction consists of a unique transaction ID, timestamp, transaction type and whatever extra data is needed for the transaction type. Transactions are only removed from the wallet when they are archived (it cannot be kept all the transactions of a wallet in the same document due to the size constraint and thus the transaction for a wallet is archived after a max size is reached and they are put them into another storage and then removed from the document).

Since there is real money involved, losing data is not an option and custom conflict resolution logic is required. The current conflict resolution logic rebuilds the wallet based on the transactions. With CRDTs the balances could probably be represented as a map of currency name and value counters, and the vouchers and transactions as sets as were presented in [10]. The Wallet service supports the following operations:

1. Purchase item (adds voucher to current vouchers set)
2. Purchase virtual currency (increases balance, current counter)
3. Consume voucher (removes voucher by adding voucher to used vauchers set)
4. Consume virtual currency (decreases balance by adding used currency count)

All of the operations add an entry to the transaction log.

3.3.1 Conflict Situations

Purchasing an item that the user should be able to purchase only once (e.g. removing ads from a game, purchasing a level package for a game) multiple times would cause problems as we would charge the item multiple times but would only be able to deliver it once.

Consuming the same voucher multiple times would cause issues if it resulted in delivering the same item multiple times (the user would have paid it only once). We could, of course, decide to take the hit and give the extra items for free.

Consuming virtual currency in a way that balance becomes negative would also cause issues unless it is decided to take the hit and round it up to 0.

3.3.2 Transactions

The transaction log needs to contain entries for all operations where real money is involved. Depending on how the transaction log is implemented (as a part of the wallet object itself or as separate document(s)), there might be a need to update more than one object atomically. If the transaction log is in separate document(s) both the wallet object and the transaction log object need to be updated, either at the same time or so that the transaction object is updated right after the wallet object.

A mathematical representation of this use case is represented below.

1. $Clients$ is the set of clients and c represents a client in the system, $c \in \mathbb{Z}_+$ and $c \in Clients$, , where $|Clients|$ is the number of clients.

2. $Balance = \{\langle Crsytals, n1\rangle, \langle Euro, n2\rangle\}$ maps each currency $curr \in \{Crystals, Euro\}$ to an amount $n1, n2 \in \mathbb{R}$. $B_{ci} \in B$ keeps the balance and $\widetilde{B}_{ci} \in B$ keeps the consumed amounts of currencies by a client c in DC i, where B denotes the set of all $Balance$ maps.

   We define the operations $\oplus$ and $\ominus$ on Balance maps $b, b' \in B$ such that: $b \oplus \langle amount \in Z, curr \in Curr\rangle = b'$ where $b'[curr] = b[curr] + amount,\quad b'[cr] = b[cr] \iff cr! = curr$ and $b \ominus (amount \in R, curr \in Curr) = b'$ where $b'[curr] = b[curr] - amount,\quad b'[cr] = b[cr] \iff cr! = curr$. We overload these operations such that they can subtract not only a tuple but a set of tuples (defined in a balance) from another balance: $b \oplus b'$ and $b \ominus b'$ where $b, b' \in B$.

3. The tuple $Voucher = \langle id, cost, spec\rangle$ defines a voucher where $id \in \mathbb{VID}$ is the voucher identifier (ID), $cost \in \mathbb{Z}_+ \times Curr$ and $spec \in Strings$ is the details of the voucher. $V_{ci} \in V$ is a multiset of vouchers of a client c kept in DC i, where V denotes the set of all vouchers. Similarly, $\widetilde{V}_{ci}$ is the multiset of consumed vouchers a client c. (Added: The cost of the voucher (Can the cost be specified in terms of different currencies?))

4. The tuple $Trans = \langle id, ts, type, args, ops\rangle$ defines a transaction where $id \in \mathbb{TID}$ is the unique transaction ID, $ts \in \mathbb{Z}_+$ is the timestamp, $type \in TTypes$ and $args \in Args$ is the data required for the transaction type. ops is a list of operations $o \in Ops = \{purchasedVouc \times V,\ purchasedVCurr \times \mathbb{Z}_+ \times Curr,$
$consumedVouc \times \mathbb{Z}_+,\ consumedVCurr \times \mathbb{Z}_+\}$ defines an operation performed in a transaction. (Added: The list of ops in the transaction.)

5. The tuple $W_{ci} = \langle B_{ci}, \widetilde{B}_{ci}, V_{ci}, \widetilde{V}_{ci}, T_c\rangle$ defines the wallet of a client c kept in DC i. A wallet keeps the purchased currencies, consumed currencies, purchased set of vouchers, consumed set of vouchers and the list of (not yet archived) transaction logs T_c of a client. $|T_c| <= MaxTSize$ since the transaction logs in a wallet should be archived when the cardinality of the transactions reaches to the max size.

6. The net balance of a client can be obtained by subtracting the consumed amounts of currencies from the balance of a client $c \in Clients$ in DC i. Equation 25 shows that the net amounts of all currencies should be non-negative.

$$\langle curr, n\rangle \in B'_{ci} = B_{ci} \ominus \widetilde{B}_{ci} \implies n \geq 0\ \forall\ curr \in Curr,\ c \in Clients,\ i \in \{1, \ldots, |DC|\} \tag{25}$$

7. The net set of vouchers of a client can be obtained by subtracting the consumed vouchers from the gained vouchers of a client $c \in Clients$ in DC i, as shown in Equation 26, where $\setminus$ is the standard multiset difference operator.

$$V'_{ci} = V_{ci} \setminus \widetilde{V}_{ci} \tag{26}$$

8. The consumed set of vouchers should be a subset of gained vouchers of a client c, as shown in Equation 27.

$$\widetilde{V}_{ci} \subseteq V_{ci} \tag{27}$$

9. An operation $o \in Ops$ maps a wallet W_{ci} to its new contents W'_{ci} $W_{ci} = \langle B_{ci}, \widetilde{B}_{ci}, V_{ci}, \widetilde{V}_{ci}, T\rangle \xrightarrow{o} W'_{ci} = \langle B'_{ci}, \widetilde{B'}_{ci}, V'_{ci}, \widetilde{V'}_{ci}, T'\rangle$ as follows:

   For purchase item operation that purchases voucher v:

   $$o = \langle purchasedVouc, v, curr\rangle \text{ where } v = \langle id, cost, spec\rangle \in V,\ curr \in Curr:$$

   Additional parameter curr (which currency to use for purchasing? Can we buy a voucher using Crystals?) Does "purchased vouc" reduces balance?

   As shown in Equation 28, the new contents of the wallet has: (i) the same balance (ii) the cost of the voucher v added to the consumed balance (iii) the purchased voucher added to the voucher set (iv) the same set of consumed vouchers and (v) the transaction logs T' that appends that purchase operation to the previous logs.

   $$B_{ci}[curr] > cost \implies \langle B_{ci}, \widetilde{B}_{ci}, V_{ci}, \widetilde{V}_{ci}, T\rangle \xrightarrow{o} \langle B_{ci}, \widetilde{B}_{ci} \oplus \{cost, curr\}, V_{ci} \cup \{v\}, \widetilde{V}_{ci}, T'\rangle$$
   $$where\ v = \langle id, \langle cost, curr\rangle, spec\rangle \in V. \tag{28}$$

   For purchase virtual currency operation that purchases an *amount* of *currency*:

   $$o = \langle purchasedVCurr, amount, currency\rangle \text{ where } amount \in \mathbb{Z}_+ \text{ and } currency \in Curr:$$

   As shown in Equation 29, the new contents of the wallet has: (i) the balance increased by the purchased amount of currency (ii) the same amount of consumed balance (iii) the same set of vouchers (iv) the same set of consumed vouchers and (v) the transaction logs T' that appends that purchase virtual currency operation to the previous logs.

   $$\langle B_{ci}, \widetilde{B}_{ci}, V_{ci}, \widetilde{V}_{ci}, T\rangle \xrightarrow{o} \langle B_{ci} \oplus \{amount, curr\}, \widetilde{B}_{ci}, V_{ci}, \widetilde{V}_{ci}, T'\rangle \tag{29}$$

   For consume voucher operation that consumes v:

   $$o = \langle consumedVouc, v\rangle \text{ where } v = \langle id, cost, spec\rangle \in V:$$

As shown in Equation 30, the new contents of the wallet has: (i) the same balance (ii) the same amount of consumed balance (iii) the same set of vouchers (iv) the set consumed vouchers together with that recently consumed voucher v and (v) the transaction logs T' that appends that consume voucher operation to the previous logs.

$$v \in V_{ci} \implies \langle B_{ci}, \widetilde{B}_{ci}, V_{ci}, \widetilde{V}_{ci}, T\rangle \overset{o}{\to} \langle B_{ci}, \widetilde{B}_{ci}, V_{ci}, \widetilde{V}_{ci} \cup \{v\}, T'\rangle \tag{30}$$

For consume virtual currency operation that consumes *amount* of *currency*:

$$o = \langle consumedVCurr, amount\rangle \text{ where } amount \in \mathbb{Z}_{+}:$$

As shown in Equation 31, the new contents of the wallet has: (i) the same balance (ii) the consumed balance increased by the consumed amount of currency (iii) the same set of vouchers (iv) the same set of consumed vouchers and (v) the transaction logs T' that appends that consume virtual currency operation to the previous logs.

$$\langle B_{ci}, \widetilde{B}_{ci}, V_{ci}, \widetilde{V}_{ci}, T\rangle \overset{o}{\to} \langle B_{ci}, \widetilde{B}_{ci} \oplus \langle amount, curr\rangle, V_{ci}, \widetilde{V}_{ci}, T'\rangle \tag{31}$$

In all these formulas, $T'_c = T_c \cup \{\langle id, ts, type, args, o\rangle\}$, assuming the operation o is the only operation performed in transaction T_c. The operations are applied in their order of appearance in the list of operations in the transaction.

4 Enterprise Applications

Trifork is a software company that has taken part in many industry projects, and have provided three unique cases from their current applications. These applications are also relevant to other domains and are as such highly relevant for challenging the requirements for CRDTs.

4.1 Shared Medicine Record (FMK)

At the surface, this is a quite simple online system: for each person, maintain a list of current treatments, which may involve one or more prescriptions, and additionally a set of events that has occurred for the given treatment, which components are defined in Table 6.

Not all treatments require prescriptions, i.e. the doctor can tell you to drink water, or take calcium tablets which you can get without a prescription, but he may make a prescription on these too. Everything prescribed will be in the system. Events are things that have happen in the real world, such as a drug being administered to a patient by a nurse, or a drug being handed out at a pharmacy.

The wide adoption of this system builds upon a successful cross-sectoral standardisation of medicine workflows and closely related concepts. The system is not an electronic health record with specialised information such as test results, measurements or the like.

One of the primary design criteria for FMK is to provide high availability. The system is in use 24x7, and currently has 40+ integrations with other healthcare systems, most of which are required to use FMK as the primary storage for relevant medicines data.

Though being simple at the high level, much of the challenge lies in making the system highly available, scalable and secure, supporting a wide range of use cases (as well as old APIs), at the same time that making sure that data flows in from many of the connected systems has some measure of consistency. In many cases data “updates” are made on the basis of a previous“query” to the system, and the system needs to have a model that captures conflicting updates. As such, this seemingly simple system ends up being surprisingly complex, especially because of the high availability requirement.

In the context of making healthcare decisions, it is much better to have some information than none. Better to have old information than none. Events that happen “outside” the system have indisputably happened, so the system needs to ingest them regardless of consistency. All this leads us down the road to a CRDT-like data model deployed on Riak (dynamo-style EC w/write-conflict capture). The central patient information data model is essentially a stateful CRDT, that exposes a semantic model for write conflicts. Ideally, there would be a replica of the entire service+dataset in each major geographical region/hospital, which is still an eventual goal. Writes should be propagated “as soon as possible”, but lack of such propagation - WAN failure - should not render the system unusable.

There is more interest in the integration of some other applications and approaches with the FMK as shown in [12, 6], which increase the relevance of the FMK in the support of the national healthcare services and the empowering of patients.

Name	Description	Type
$Patients_d$	It is the set of patients part of the system (FMK) as seen by DC d and p represents one of those patients as seen by DC d ($p \in Patients$), with $\|Patients_d\|$ corresponding to the number of patients as seen by DC d.	$p \in \mathbb{Z}_+$
$Treatments_{dp}$	It is the set of treatments for patient p as seen by DC d, where $\|Treatments_{dp}\|$ corresponds to the number of treatments already registered for patient p as seen by DC d.	
$Prescriptions_{dpt}$	It is the set of prescriptions for treatment t and patient p as seen by DC, where $\|Prescriptions_{dpt}\|$ corresponds to the number of prescriptions for treatment t of patient p as seen by DC d.	
$Prescriptions_{dptr}$	It is the prescription r in treatment t for patient p as seen by DC d ($Prescriptions_{dptr} \in Prescriptions_{dpt}$ and $r \in \{1, \ldots, \|Prescriptions_{dpt}\|\}$).	$r \in \mathbb{Z}_+$
$Events_{dpt}$	It is the set of prescriptions for treatment t and patient p as seen by DC d ($e \in Events_{dpt}$), where $\|Eventsdpt\|$ corresponds to the number of events for treatment t of patient p as seen by DC d.	
$Events_{dpte}$	It is the prescription e for treatment t of patient p as seen by DC d ($Events_{dpte} \in Events_{dpt}$ and $e \in \{1, \ldots, \|Events_{dpt}\|\}$).	$e \in \mathbb{Z}_+$
$Treatments_{dpt}$	It is the treatment t for patient p as seen by DC d ($Treatments_{dpt} \in Treatments_{dp}$ and $t \in \{1, \ldots, \|Treatments_{dp}\|\}$). It is also a tuple composed of prescriptions ($Prescriptions_{dpt}$) and events ($Events_{dpt}$) part of the treatment t for patient p as seen by DC d.	$t \in \mathbb{Z}_+$

Table 6: FMK Constants and Variables.

4.1.1 Network Topology and Architecture

The system is made up of geographically separated DCs, set up in master-master replication mode, so any DC can handle any request. The client systems are systems providers for General Practitioners (GPs), pharmacies and hospitals as well as a web based system that provides citizens access and acts as a backup for the professional systems. Each client has an affinity to a given primary DC, so all requests from a given client use only one DC, as long as it is available.

4.1.2 Conflict Situations

Because of the asynchronous client system interfaces, and distributed DCs, two doctors can prescribe conflicting medicines to the same patient simultaneously. A real-life example of this is right after a patient is discharged from hospital and visits his GP. The medicines that a patient was prescribed in the hospital is sometimes carried over from the hospital patient journal to FMK after his discharge, and can coincide with the prescription of new medicine by a GP. Because the system is EC, it is not always visible, that all updates have not yet propagated throughout. This means that conflicts can be detected after the conflicting changes were made. "Conflicting medicine" may be multiple prescriptions of drugs containing the same active substance, or two drugs which interact poorly. Optimally, a doctor making or adjusting a prescription has full overview of the patient's existing prescriptions when he/she does so.

4.1.3 Format Representation (work in progress)

- **Create Treatment**: when creating a new treatment for a patient p through DC d the new treatment will be part of the list of treatments for that patient $Treatments_{dp}$, as shown in Equation 32. Also the patient must already exist.

$$
\begin{aligned}
Treatments_{dp} &= Treatments_{dp} \cup Treatments_{dpt} \\
&\text{such that } d \in \{1, \ldots, |DC|\}, p \in \{1, \ldots, |Patients_d|\}, t \notin Treatments_{dp}
\end{aligned}
\tag{32}
$$

- **Add Prescription**: when creating a new prescription r for treatment t of patient p through DC d the new prescription will be part of the list of prescriptions for such treatment ($Prescriptions_{dpt}$)

, as shown in Equation 33. Also the patient and treatment must already exist.

$$\begin{aligned}Prescriptions_{dpt} = Prescriptions_{dpt} \cup Prescription_{dptr} \\ \text{such that } d \in \{1,\ldots,|DC|\}, p \in \{1,\ldots,|Patients_d|\}, \\ t \in \{1,\ldots,|Treatments_{dp}|\}, r \notin Prescription_{dpt}\end{aligned} \tag{33}$$

- **Add Event**: when creating a new event r for treatment t of patient p through DC d the new event will be part of the list of events for such treatment ($Events_{dpt}$) , as shown in Equation 34. Also the patient and treatment must already exist.

$$\begin{aligned}Events_{dpt} = Events_{dpt} \cup Events_{dpte} \text{ such that } d \in \{1,\ldots,|DC|\}, p \in \{1,\ldots,|Patients_d|\}, \\ t \in \{1,\ldots,|Treatments_{dp}|\} e \notin Events_{dpt}\end{aligned} \tag{34}$$

Given that prescriptions and events corresponds to things that have already happened then they cannot be removed from a patient treatment, so no need for delete operations.

The record for a patient is said to be out of sync if there is a discrepancy between the treatments for that patient as seen by different DCs in the system:

1. Different treatment(s) in either or both of the records seen by any two DCs d and q. Equation 35 states that for a patient p exists a treatment, t, in his/her record presented by DC d that does not exist in the record for the same person shown by DC q.

$$\exists\ t \in 1,\ldots,|Treatments_{dp}|,\ Treatments_{dpt} \notin Treatments_{qp} \tag{35}$$

2. Differences within the same treatment for the same person are shown between any two DC in the system:

   (a) Difference(s) between the prescriptions within a treatment t for a patient p. Equation 36 states that for a treatment t of a patient p exists a prescription, r, in his/her record presented by DC d that does not exist in the record for the same person shown by DC q.

$$\exists\ r \in 1,\ldots,|Prescriptions_{dp}|,\ Prescriptions_{dptr} \notin Prescriptions_{qpt} \tag{36}$$

   (b) Difference(s) between the events within a treatment t for a patient p. Equation 37 states that for a treatment t of a patient p exists an event, e, in his/her record presented by DC d that does not exist in the record for the same person shown by DC q.

$$\exists\ e \in 1,\ldots,|Events_{dp}|,\ Events_{dpte} \notin Events_{qpt} \tag{37}$$

So a patient record is out of sync if any combination of the cases above appear for a patient between different DCs.

4.2 Festival

The Trifrok Festival application is an existing App developed for Android and iOS, Figure 2. It allows festival participants to see the concert schedule and other centrally updated information as well as distribution of user-created content. The application has been operational for years while each year new features have been added. The specific use case we are addressing here is the ability to conduct polls where participants can vote for a concert. The challenge is that we cannot see if we receive a vote twice, we don't know who the vote came from, and we do not have a reliable network with a DC in the middle.

Massive events like conferences, sport events, and music festivals encounters may saturate the mobile bandwidth which would result in loss of cellular radio connectivity. Based on Trifork Relai local data are updated via not only cellular radio connectivity but also Blue-tooth and WiFi Direct with other devices in a peer-to-peer fashion. So although you may not be able to connect to a DC you can still post your votes to peer devices just as you can get more up to date data from these.

Having a normal counter where you add one or more at the time will obviously not work in this scenario as we don't have a central database, we have no means to check how many times you are voting,

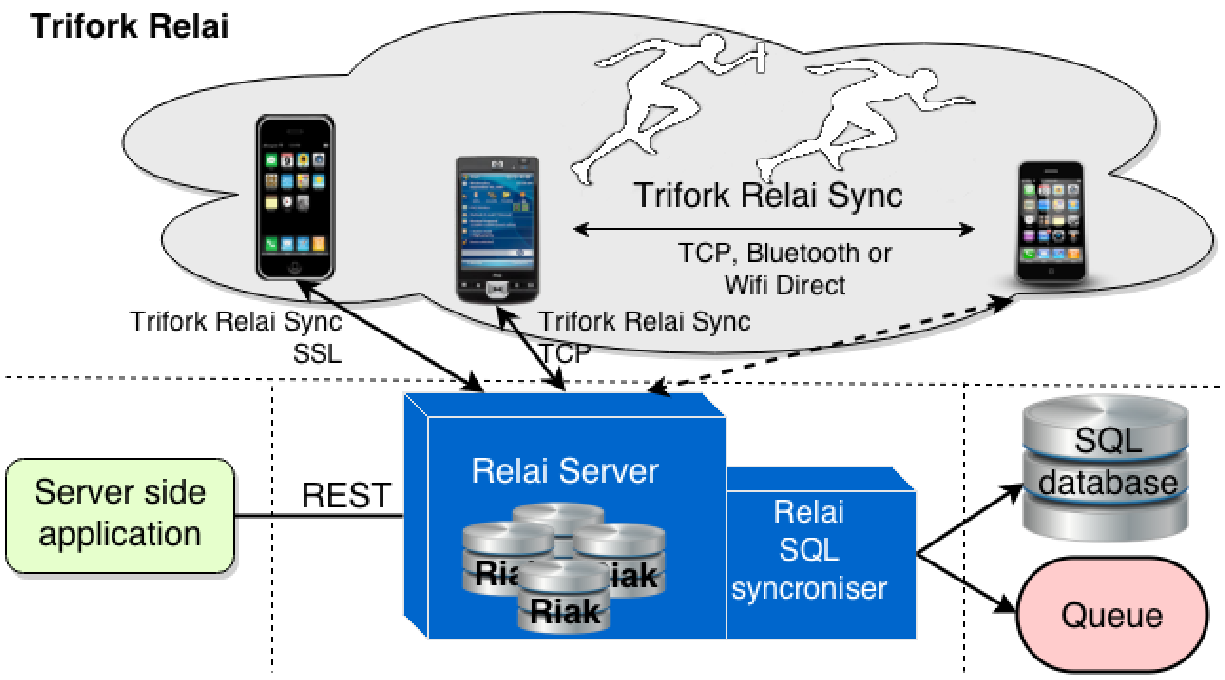

Figure 2: Overview of current Festival implementation which uses Trifork's Relai.

nor do we have any means to see how large a percentage of possible votes have been given. This raises a new dilemma where the probabilistic counter comes in handy.

Each device will hold a bit array for voting bad and for voting good for each concert, i.e. when a vote is cast as "bad" a random number is generated on the device and the corresponding bit in the "bad" array is set. The same goes for the "good" votes, but note that each possible candidate (bad or good in this case) must have their own bit array. Now this array can be spread to peer devices where it is added to other device's bit arrays as a simple AND operation. At any device you can now see the total number of votes for bad and good by calculating the number of votes that are most likely with number of set bits compared to the number of still unset bits. This value will not change if an array is added several times, in other words it is idempotent.

One consistency issue that is left unresolved is that you cannot see that few or many votes are missing. Obviously the precision of the count must be obtained by sizing the array towards the total number of votes cast. Another possible inconsistency is that segregated groups of devices can show uncorrelated results. This can happen in a number of situations which are unlikely but possible, i.e. lets say no cellular radio network is available and one group are now only networking using WiFi Direct and another group are only networking using Blue-tooth and no device is bridging the two means of networking. Then each group will have their own voting polls.

We have a number of unintended side effects. For a music festival this is acceptable, while for an App for parliament this would not be acceptable. In the implementation each device can only vote for each concert once and you cannot alter your vote after it has been given. If you have more devices you also have more votes. If you uninstall and reinstall the App you will be able to vote again for the same concert.

4.2.1 Mark-Counter

A Mark-Counter is composed of an array of n bits. A Mark-Counter B is composed of n bits each represented by $B_i \in \{0,1\} \forall i \in \{1,\dots,n\}$. A^n represents the group of all the arrays of bits of size n ($|B| = n$), where n is also the number of bits in a Mark-Counter.

Bitwise OR operator is represented as — in this document.

Operations:

- Set a bit in a Mark-Counter: a bit may change individually from 0 to 1 but may not be reset back to zero by this operator.

- Merge Mark-Counters (bitwise OR operator, —): given two Mark-Counters $B, C \in A^n$, their merger is defined as $D = \{D_i = B_i|C_i, i \in \{1, \ldots, n\}\}$.

  If both arrays have different sizes, i.e. $|B|$ and $|C|$, then $D = \{D_i = B_i|C_i, i \in \{1, \ldots, \max\{|B|, |C|\}\}$ with $B_j = 0\ \forall\ j > |B|, C_k = 0\ \forall\ k > |C|\}$ and the size of D would be $|D| = \max\{|B|, |C|\}$.

- Count: given a Mark-Counter B the count corresponds to the number of bits that have been set to 1, $\sum_{i=1}^{|B|} B_i$. Similarly, once it is know the number of 1s it is also known the number of zeros for a given array size.

A Mark-Counter is a CRDT: A Mark-Counter is a simple data structure which complies with the requirements to be a CRDT, as shown below.

- **Commutative**. Given two Mark-Counts B and C with n bits each where a bit index is presented by i and its value in the Mark-Counter by B_i and C_i respectively the the merging of the corresponding bit at position i provide the same result irrespective of the order the bit in each Mark-Counter is executed, as shown by, Equation 38.

$$B_i|C_i = C_i|B_i \tag{38}$$

  The prove is shown on the Table 7 where the two first columns contain the input Mark-Counters and the last two columns the different order they may be computed which provide the same results.

Values		Results	
B_i	C_i	$B_i\|C_i$	$C_i\|B_i$
0	0	0	0
1	0	1	1
0	1	1	1
1	1	1	1

Table 7: Mark-Counter commutative prove, $i \in \{1, \ldots, n\}$.

- **Associative**, Equation 39.

$$(B_i|C_i)|D_i = B_i|(C_i|D_i) \tag{39}$$

  The prove is shown on the Table 8.

Values			Intermediate		Results	
B_i	C_i	D_i	$B_i\|C_i$	$C_i\|D_i$	$(B_i\|C_i)\|D_i$	$B_i\|(C_i)\|D_i)$
0	0	0	0	0	0	0
1	0	0	1	0	1	1
0	1	0	1	1	1	1
1	1	0	1	1	1	1
0	0	1	0	1	1	1
1	0	1	1	1	1	1
0	1	1	1	1	1	1
1	1	1	1	1	1	1

Table 8: Mark-Counter associative prove, $i \in \{1, \ldots, n\}$.

- **Idempotent**, Equation 40.

$$B_i|B_i = B_i \tag{40}$$

  The prove is shown on the Table 9.

Value	Result
B_i	$B_i \mid B_i$
0	0
1	1

Table 9: Mark-Counter idempotent prove, $i \in \{1, \dots, n\}$.

Name	Description	Type
$People$	It is the set of people attending the "Festival" and p represent one of those people ($p \in People$), with $\|People\|$ representing the number of people attending and the size of the array of bits.	$p \in \mathbb{Z}_+$
n_d	It refers to the highest attendee's index to the "Festival" that the device d has information about, $d \in \{1, \dots, \|People\|\}$. Also $1 \leq n_d \leq \|People\| \; \forall \; d \in \{1, \dots, \|People\|\}$.	$\mathbb{Z}_+$
G_d	It is the array of good votes known by the device d from device d and other devices. G_d is an array of bit of size of at least n_d, with each bit represent a device. G_d is a Mark-Counter where a bit is set to 1 if the corresponding attendee, represented by that bit, is set to 1 to state the vote of the attendee as good.	$\mathbb{Z}_+$
G_{da}	It is the value in the array of good votes available at devise d for attendee a, presented in Expression 41. Also it can be said to be the bit a in the array of bits G_d, $a \in \{1, \dots, \|People\|\}$. $$0 \leq G_{ka} \leq 1 \; \forall k, a \in \{1, \dots, \|People\|\} \quad (41)$$	$\mathbb{Z}_+$
B_d	It is the array of bad votes known by device d from device d and other devices. B_d is an array of bit of size of at least n_d, with each bit represent a device. B_d is a Mark-Counter where a bit is set to 1 if the corresponding attendee, represented by that bit, is set to 1 to state the vote of the attendee as bad.	$\mathbb{Z}_+$
B_{da}	It is the value in the array of bad votes seen by device d for attendee a, B_{da}, presented in Expression 42. Also it can be said to be the bit a in the array of bits B_d, $a \in \{1, \dots, \|People\|\}$. $$0 \leq B_{da} \leq 1 \; \forall d, a \in \{1, \dots, \|People\|\} \quad (42)$$	$\mathbb{Z}_+$
$numGood_d$	It is the number of good votes received by the attendee d device, as shown by Equation 45.	$\mathbb{Z}_+$
$numGood$	It represents the total number of attendees that voted good, as shown in Equations 43 and 44. It is considered that if $n_d < \|People\|$ then $G_{da} = B_{da} = 0 \; \forall \; a > n_d$.	$\mathbb{Z}_+$
$numBad_d$	It is the number of bad votes received by the attendee d device, as shown by Equation 48.	$\mathbb{Z}_+$
$numBad$	It represents the total number of attendees that voted bad, as shown in Equations 46 and 47.	$\mathbb{Z}_+$
$numAttendees_d$	It represents the total number of attendees that voted as seen by device/attendee d, as shown in Equation 49.	$\mathbb{Z}_+$
$numAttendees$	It represents the total number of attendees that voted, as shown in Equation 50.	$\mathbb{Z}_+$

Table 10: Festival Constants and Variables.

4.2.2 Festival use case with Mark-Counters

Consider a "Festival" which it is composed of many acts/events. We start by considering the case of a "Festival" composed only of one act that later will be extended to many. To simplify, without losing generality, it is considered the case of an event in a theatre and the constants and variables used are presented in Table 10.

$$\delta_d^{good} = \begin{cases} 1 & if \sum_{a \in People} G_{da} > 0 \\ 0 & otherwise \end{cases} \tag{43}$$

$$numGood = \sum_{d \in People} \delta_d^{good} \tag{44}$$

$$numGood_d = \sum_{a=1}^{n_d} G_{da} \ \forall \ d \in \{1, \ldots, |People|\} \tag{45}$$

$$\delta_d^{bad} = \begin{cases} 1 & if \sum_{d \in People} B_{da} > 0 \\ 0 & otherwise \end{cases} \tag{46}$$

$$numBad = \sum_{d \in People} \delta_d^{bad} \tag{47}$$

$$numBad_d = \sum_{a=1}^{n_d} B_{da} \ \forall \ d \in \{1, \ldots, |People|\} \tag{48}$$

$$numAttendees_d = \sum_{a=1}^{n_d} (G_{da} + B_{da}) \ \forall \ d \in \{1, \ldots, |People|\} \tag{49}$$

$$numAttendees = numGood + numBad \tag{50}$$

An attendee may only vote once as expressed in Inequality 51, but he/she may not vote at all.

$$G_{da} + B_{da} < 2 \ \forall \ d \in \{1, \ldots, |People|\}, a \in \{1, \ldots, n_k\} \tag{51}$$

A measure of the coherence for the copies between an attendee a and another attendee j can be obtained by calculating the number of bits where both versions of voting differ as expressed in Equation 52. All the devices are in sync if $\triangle numAttendees_{dj} = 0 \ \forall \ d, j \in \{1, \ldots, |People|\}$.

$$\triangle numAttendees_{dj} = \sum_{i \in People} ((G_{di} + G_{ji})\%2 + (B_{dj} + B_{ji})\%2) \forall \ d, j \in \{1, \ldots, |People|\} \tag{52}$$

This can be extended to multiple events/acts by introducing a new index that represent each of these events/acts.

The problem with the real Festival use case is that not all devices may know about all the attendees and even if they did then the size of the bit arrays may be too big to be efficient their use, so Trifork come with the idea of using Statistical Mark-Counter. which are presented in Section 4.2.3. In Trifork's Festival the counters are reduced in size by using statistics, and each bit is randomly assigned to a customer.

4.2.3 Statistical Mark-Counter

To reduce the amount of data transferred between devices and for cases where not all devises may know the total number of attendees it could be used a probabilistic approach. The current Trifork's Festival implementation already uses an statistical Mark-Counter, where the index of an attendee is calculated randomly for a pre-defined size of the poll, $n_d = n \leq |People| \ \forall \ d \in \{1, \ldots, n\}$. This means that different attendees may be assigned the same position in the poll, so potentially their votes would equate to a single vote, if they vote equally. So given this the Inequality 53 would not be complied with so it would needs to be removed from the representation to 46.

$$G_{da} + B_{da} \leq 2 \ \forall \ d, a \in \{1, \ldots, n\} \tag{53}$$

The size of the array of bits depends of the quality expected from the results extrapolated, whereas larger sizes generally lead to increased precision, e.g. a size equal to the number of attendees will provide the maximum precision. In practice, the size used in a study is determined based on the cost of data collection, storage, transmission, processing, and the need to have sufficient statistical power. Sizes may be chosen in several different ways such as target for the power and target variance of a statistical test.

5 Business to Business (B2B)

This application was built from the ground up for a large clothes manufacturer, who sells boxes of clothes to thousands of stores in 30+ markets. It replaces a manual process, where travelling salesmen visited the stores and presented physical clothing from their sample collections.

The B2B order application enables the clients, e.g. shop employees, to see a catalogue of upcoming clothes on a tablet device, and place orders on boxes for future delivery, shown in Table 11. This functionality is made available for the same shops to shop staff as well as for shop managers, managers of chains of shops or managers of entire markets.

Name	Description	Type
$Business_d$	It is the set of businesses as seen by the DC d and b represents one of those businesses, $b \in Businesses$.	
$Clothes_{db}$	It is the set of clothes for business b as seen by the DC d and l represents one of those clothes, $l \in Clothes_b$.	
$Cost_{dbl}$	It is the cost of a box of clothes l from business b as seen by the DC d.	$\mathbb{R}_+$
$Capacity_{dbl}$	It is the maximum quantity of available boxes of clothes l as seen by the DC d.	$\mathbb{Z}_+$
$Clients_{db}$	It is the set of clients for business b as seen by the DC d and c represents one of those clients, $c \in Clients_b$.	
$Credit_{dbc}$	It is the credit of client c from business b as seen by the DC d.	$\mathbb{R}$
$Boxes_{dblc}$	It is the number of boxes of clothes l from business b ordered by client c as seen by the DC d.	$\mathbb{Z}_+$

Table 11: B2B Constants and Variables.

Constraints for this problem:
It is not allowed to order more boxes for a particular clothe c from a business b between all the clients of that business than those available $Capacity_{dbl}$, but less may be ordered as expressed in Inequality 54.

$$Caplcity_{dbl} \geq \sum_{c \in Clients_{db}} Boxes_{dblc} \tag{54}$$

It is not allowed to expend by any of the clients in orders more that their credit from business b as expressed by Inequality 55.

$$Credit_{dbc} \geq \sum_{l \in Clothes_{db}} (Boxes_{dblc} * Cost_{dbl}) \tag{55}$$

Orders are modelled as a State-based increment-only Counter (G-set) [9], which is updated by adding event objects.

The tablets can operate off-line and only need to be online for getting i.e. new catalogue replicated down and for posting orders towards the server. It is expected that stores cannot be spread over many DCs. More DCs can be used and in this case each DC must be aware of in which DC each shop's data is located.

The tablets can remain off-line for any length of time. This effectively means that there will be issues with double orders, out of stock, not current catalogue, and other conflicts. The automated system is not intended for handling these conflicts while it will attempt to detect conflicts and potential conflicts. These will then be brought to the attention of manufacturer's customer support.

Cancellation and adjustment of orders are not offered via the tablet solution. These will have to be dealt with by the manufacturers customer support. An overall view of the system can be seen in Figure 3.

6 Conclusion

We have seen databases residing in one DC, in replicated DCs, on mobile phones, and distributed across all with potential eventual consistency. But we have also seen how data is present at the point of entry and possible believed by the user to be persistent in a "central" although maybe never delivered. These scenarios represents the CRDTs at

- Server side

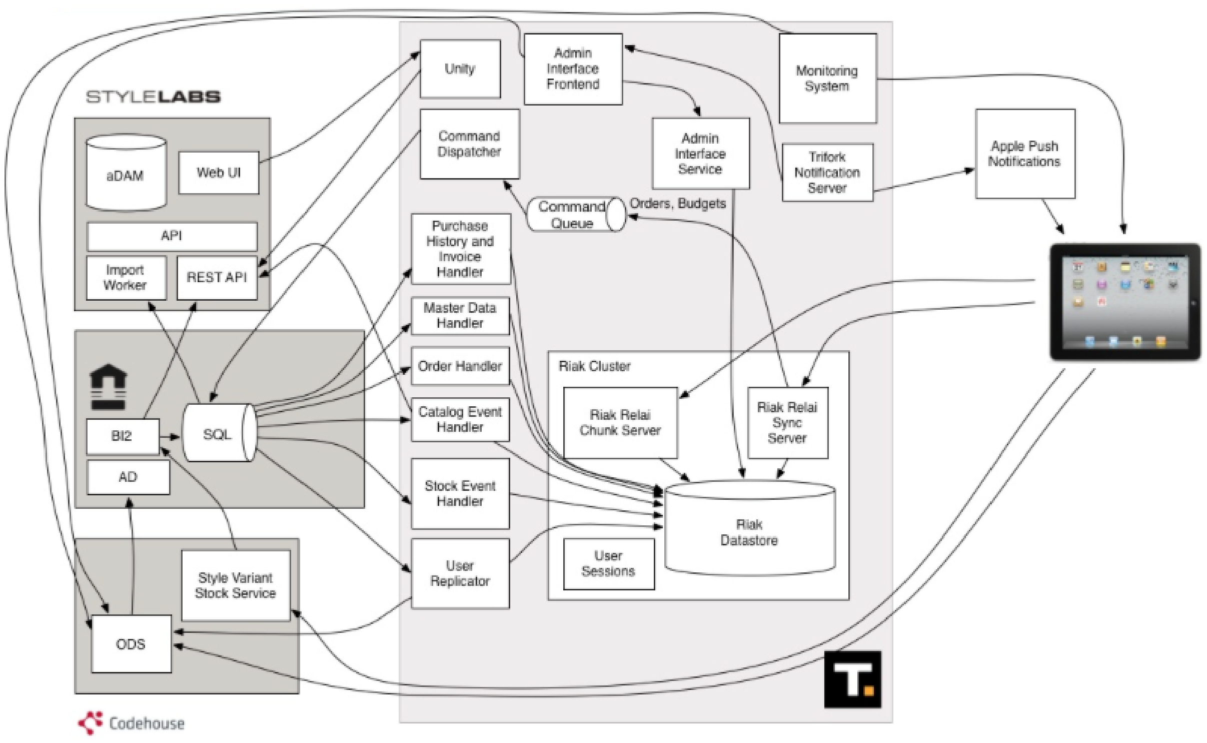

Figure 3: B2B eCommerce Architecture (P2tF).

- Points-of-presence
- Edge of network

The attributes of the data in process

- Object composition and transaction
- Divergence and Quality of data
- Security

Some of the requirements we have looked at will be supported early and can be verified in the basic programming model. Others are more comprehensive and cannot be expected to be successfully verified until later. Eventually some requirements will not be satisfied as they are not feasible in an automatic context.

- Basic programming model
- Extended programming model
- Final programming model

All of the examples use as input to the requirement gathering are from the real world and they amongst others represent a world wide hundred billion Euro electronic entertainment industry and the Health Care Sector taking a two digit percentile of Western State Budgets. The present IT Industry have very little support for this kind of applications and as we have seen the problems addressed in the use cases are not supported in classic database implementations.

7 Acknowledgements

This has been created as part of the European research project SyncFree.

References

[1] D. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, Feb 2012.

[2] C. Baquero and F. Moura. Specification of convergent abstract data types for autonomous mobile computing. Technical report, Departamento de Informática, Universidade do Minho, 1997.

[3] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[4] A. Deftu and J. Griebsch. A scalable conflict-free replicated set data type. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, pages 186–195, Washington, DC, USA, 2013. IEEE Computer Society.

[5] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[6] K. M. Hansen, M. Ingstrup, M. Kyng, and J. W. Olsen. Towards a software ecosystem of health- care services. In *Infrastructures for Healthcare: Global Healthcare: Proceedings of the 3rd International Workshop 2011*, pages 27–36, 2011.

[7] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005.

[8] M. Shapiro. Optimistic replication and resolution. In M. T. Özsu and L. Liu, editors, *Encyclopedia of Database Systems (online and print)*. Springer, October 2009. http://www.springer.com/computer/database+management+

[9] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. Rapport de recherche RR-7506, INRIA, Jan. 2011. Printed.

[10] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976, pages 386–400, Grenoble, France, Oct. 2011.

[11] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[12] S. Urazimbetova. A case study - on patient empowerment and integration of telemedicine to national healthcare services. In *International Conference on Health Informatics*, Vilamoura, Algarve, Portugal, Februery 2012. DOI link: The full text of this paper is only available to INSTICC members.

[13] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009.

A TLA+ representation of Use Cases

A.1 Advertisement Counter (AdCounter)

```
------------------------------- MODULE AdCounterState_v2 -------------------------------

EXTENDS Naturals, GCounters

CONSTANTS
    DV, \* Set of all devices.
    DC, \* Set of all data centers.
    AD, \* Set of all ads.
    maxTotalViews, \* Maximum number of times an ad should be shown.
    maxTotalViewsPerDC, \* Maximum number of times an ad should be shown in a data center.
                        \* In this specification this partition is fixed.
    maxTotalViewsPerDevice, \* Maximum number of times an ad should be shown in a device.
    deviceAssignment \* Assignment of each device to a data center.
                     \* In this specification this assignment is fixed.
(*SumAll(func) ==
    LET sum[p \in func] ==
        IF func\{p} = {} THEN p[2] ELSE p[2] + sum[func\{p}]
    IN sum[func] *)

RECURSIVE SumAll(_)
\* Auxiliary operation that given a function f:X -> Nat
\* returns the summation of all numbers in the range of f.
SumAll(f) ==
    IF DOMAIN f = {} THEN 0
    ELSE LET v == CHOOSE x \in DOMAIN f : TRUE
         IN f[v] + SumAll([a \in (DOMAIN f) \ {v} |-> f[a]])


ASSUME maxTotalViews \in [AD -> Nat]
ASSUME maxTotalViewsPerDevice \in [AD -> [DV -> Nat]]
ASSUME deviceAssignment \in [DV -> DC]
ASSUME maxTotalViewsPerDC \in [DC -> [AD -> Nat]]
 /\ \A a \in AD : SumAll([d \in DC |-> maxTotalViewsPerDC[d][a]]) = maxTotalViews[a]

VARIABLES configuration

----------------------------------------------------------------------------------------

\* Record that represents the local state of a data center.
\* Fields maxViews and devices are constant in the current specification,
\* but the plan is to add operations for moving devices and transfer view rights
\* between data centers.
State ==
    [   devices: SUBSET DV, \* Set of devices assigned to the data center.
        ads: SUBSET AD, \* Set of devices assigned to the shown on that data center
        (* should be maxViews: [ads-> Nat] *)
        maxViews: [AD -> Nat], \* Maximum number of times an ad should be shown in
                               \* the data center. Used to transfer view rights.
        (* should be maxViews: [ads-> GCounter(DC)] *)
        views: [AD -> GCounter(DC)], \* G-Counter with overall views for each ad.
        (* should be viewsPerDevice: [ads-> [devices -> Nat]] *)
        viewsPerDevice: [AD -> [DV -> Nat]]] \* Number of views of each ad by a device.
                                             \* Note: Because devices are assigned to a single
                                             \* data center it is suficiente to keep the value
                                             \* of the local counter.

TypeInvariant == configuration \in [DC -> State]

Init ==
    /\ TypeInvariant
    /\ configuration =
        [d \in DC |-> [devices |-> {g \in DV : deviceAssignment[g] = d},
                       ads |-> {a \in AD : maxTotalViewsPerDC[d][a] > 0},
                       maxViews |-> [a \in AD |-> maxTotalViewsPerDC[d][a]],
                       views |-> [a \in AD |-> GCounterInit(DC)],
                       viewsPerDevice |-> [a \in AD |-> [g \in DV |-> 0]]]]

----------------------------------------------------------------------------------------

\* Local operation at data center d that represents a visualisation of
\* the advert a in device g at time t.
\* Pre:  - Device g is assigned to data center d.
\*       - The data center view limit for ad a is not exceeded.
\*       - The device view limit for ad a in device g is not exceeded.
\* Post: - The data center local state is updated, in particular,
\*       - the views G-Counter and viewsPerDevice is incremented by one.
Inc(d, a, g) ==
    LET state == configuration[d]
        gc == state.views[a]
        new_state ==
            [devices |-> state.devices,
             ads |-> state.ads,
             maxViews |-> state.maxViews,
             views |-> [state.views EXCEPT ![a] = GCounterInc(gc,d)],
             viewsPerDevice |-> [state.viewsPerDevice EXCEPT ![a][g] = @+1]]
    IN /\ g \in state.devices
       /\ GCounterValueAt(gc,d) < state.maxViews[a]
       /\ state.viewsPerDevice[a][g] < maxTotalViewsPerDevice[a][g]
       /\ configuration' = [configuration EXCEPT ![d] = new_state]

Merge(d1, d2) ==
    LET state1 == configuration[d1]
        state2 == configuration[d2]
        new_state1 ==
            [devices |-> state1.devices,
             ads |-> state1.ads,
             maxViews |-> state1.maxViews,
             views |-> [a \in AD |->
                    IF a \in state1.ads
                    THEN GCounterMerge(state1.views[a],state2.views[a])
                    ELSE 0],
             viewsPerDevice |-> state1.viewsPerDevice]

  IN configuration' = [configuration EXCEPT ![d1] = new_state1]

\* Operation for consulting the number of views of ad a.
Views(d,a) == SumAll(configuration[d].views[a])

----------------------------------------------------------------------------------------

Consistency ==
    \* The ad views do not exceed the total views limite.
    /\ \A d \in DC, a \in AD : Views(d,a) <= maxTotalViews[a]

    \* The views local to a data center do not exceed limite for that data center.
    /\ \A d \in DC, a \in AD : configuration[d].views[a][d] <= maxTotalViewsPerDC[a][d]

    \* A data center only keeps the views of ads assigned to it.
    \* Needed because it is not possible to define partial functions in TLA.
    /\ \A d \in DC, a \in AD : a \notin configuration[d].ads => Views(d,a) = 0

    \* The ad views of a device do not exceed the views limite for that device.
    /\ \A d \in DC, a \in AD, g \in DV :
            configuration[d].viewsPerDevice[a][g] <= maxTotalViewsPerDevice[a][g]

    \* A data center only keeps the views of devices assigned to it.
    \* Needed because it is not possible to define partial functions in TLA.
    /\ \A d \in DC, a \in AD, g \in DV :
        /\ g \notin configuration[d].devices => configuration[d].viewsPerDevice[a][g] = 0
        /\ a \notin configuration[d].ads => configuration[d].viewsPerDevice[a][g] = 0

    \* The ad views by devices matches the total ad views.
    /\ \A d \in DC, a \in AD : SumAll(configuration[d].viewsPerDevice[a]) = Views(d,a)

    \* GCounter property:
    \* The local value of a gcounter has to be greater or equal to the value in other.
    /\ \A d1 \in DC, d2 \in DC , a \in AD:
            configuration[d1].views[a][d1] >= configuration[d2].views[a][d1]

Next == \E d1 \in DC, d2 \in DC, a \in AD, g \in DV : Inc(d1,a,g) \/ Merge(d1,d2)

vars == <<configuration>>
-----------------------------------------------------------------------------
Spec == Init /\ [][Next]_vars
-----------------------------------------------------------------------------
THEOREM Spec => TypeInvariant \*/\ Consistency *)

=============================================================================
\* Modification History
\* Last modified Mon Sep 22 12:41:34 WEST 2014 by carlaferreira
\* Created Wed Sep 10 22:06:30 CEST 2014 by carlaferreira
```

A.2 Leader Board

```
--------------------------- MODULE leaderBoard_v2 ---------------------------
EXTENDS Naturals, TLC, FiniteSets
VARIABLES scores, vecclc
CONSTANTS DC, Games, Players, PlayersByGame

ASSUME PlayersByGame \in [Games -> SUBSET Players]
-----------------------------------------------------------------------------
\* Function for converting Game-Player information function to tuple form
MapProduct(map) ==
    LET mp[s \in SUBSET DOMAIN map] == IF s={} THEN {} ELSE LET y == CHOOSE x \in s: TRUE IN ({y} \X map[y])
    IN mp[DOMAIN map]

GamePlayers == MapProduct(PlayersByGame)

\* scores is a function from Data Centers to the games to the players playing this game to the Nat
\* vecclc is a ghost variable keeping vector clock for each player in each game for each data centerr
TypeInv == /\ scores \in [DC ->[MapProduct(PlayersByGame) -> Nat]]
           /\ vecclc \in [GamePlayers -> [DC -> [DC -> Nat]]]

\* Operator that returns the set that contains the players who lead in the game g according to data center d
```

```
Leaders(d,g) == {x \in PlayersByGame[g] : \A y \in PlayersByGame[g]: scores[d][<<g,x>>] >= scores[d][<<g,y>>

\* Operator for finding set of players who ranks ith in game g according to the data center d
Rank(d,g,i) ==
    LET RankSets[j \in 1..i] == IF j=1 THEN Leaders(d,g)
                                ELSE LET remaining ==  PlayersByGame[g] \ RankSets[j-1]
                                     IN RankSets[j-1] \cup {x \in remaining : \A y \in remaining: scores[d][
    IN  IF i =1 THEN Leaders(d,g) ELSE RankSets[i] \ RankSets[i-1]



\* Initialize each field of every variable to zero
Init == /\ scores = [d \in DC |-> [ a \in GamePlayers |-> 0]]
        /\ vecclc = [g \in GamePlayers |-> [d \in DC |-> [dc \in DC |-> 0]]]

\* An operation that updates the score of a user in a particular game and data center
(* PRE: - p must be a player of game g
        - new score entered must be greater than the current score of the user
        - number of updates for this player must be within natural numbers limit
   POST: - update field of the scores variable corresponding to the player p for game g in data center d
        - increment d field of the vector clock of the player p in game g for the data center d by 1 *)
UpdateScore(dc,g,p,scr) == /\ <<g,p>> \in GamePlayers
                           /\ scr > scores[dc][<<g,p>>]
                           /\ scores' = [scores EXCEPT ! [dc][<<g,p>>] = scr]
                           /\ vecclc[<<g,p>>][dc][dc]+1 \in Nat
                           /\ vecclc' = [vecclc EXCEPT ! [<<g,p>>][dc][dc] = vecclc[<<g,p>>][dc][dc]+1]

\* A helper function for finding maximum of two numerals
Max(n1,n2) == IF n1 >= n2 THEN n1 ELSE n2

\* An operation that merges the score of player p in game g in data center d1 with the same field in data ce
(* PRE: - p must be a player of game g
        - score of p in data center d2 for game g must be greater than the corresponding score in data cente
   POST:- update the field of the scores' variable corresponding to player p for game g in data center d1 by
        the scores kept in d1 and d2
        - merge the vector clock of player p for game g in data center d with the vector clock of the same u
        in data center d2 *)
Merge(d1,d2,g,p) == /\ <<g,p>> \in GamePlayers
                    /\ scores[d2][<<g,p>>] > scores[d1][<<g,p>>]
                    /\ scores' = [scores EXCEPT ! [d1][<<g,p>>] = scores[d2][<<g,p>>] ]
                    /\ LET new_vc == [d \in DC |-> Max(vecclc[<<g,p>>][d1][d],vecclc[<<g,p>>][d2][d]) ] IN
                    (*/\ LET a== <<"MERGE",d1,g,1,Rank(d1,g,1),scores>> IN Print(a,TRUE)
                    /\ LET a== <<"MERGE",d1,g,2,Rank(d1,g,2),scores>> IN Print(a,TRUE)
                    /\ LET a== <<"MERGE",d1,g,3,Rank(d1,g,3),scores>> IN Print(a,TRUE)*)



Next == \E d \in DC, d2 \in DC, g \in Games, p \in Players, scr \in Nat: (UpdateScore(d,g,p,scr) \/ Merge(d,

Spec == Init /\ [][Next]_<<scores,vecclc>>

\* Eventual Consistency invariant stating that scores variable eventually converges
Convergence == <> \A d1 \in DC, d2 \in DC, g \in Games, p \in Players: <<g,p>> \in GamePlayers => scores[d1]
\* if the vector clock for the all players playing game g in data center d1 is <= vector clocks of the same
\* d2 then Leader's score for g in d1 must be >= Leader's score for g in d2
MonotonicLeader == \A d1 \in DC, d2 \in DC, g \in Games:
    (\A p \in Players: <<g,p>> \in GamePlayers => (\A d \in DC: vecclc[<<g,p>>][d1][d] <= vecclc[<<g,p>>][d2

=============================================================================
\* Modification History
\* Last modified Tue Oct 21 03:38:40 EEST 2014 by Suha
\* Created Tue Sep 30 20:42:31 EEST 2014 by Suha
```

A.3 Virtual Wallet

```
------------------------------ MODULE walletv4 ------------------------------
EXTENDS Naturals, Sequences, TLC
VARIABLES wallets
CONSTANTS Replicas, V1Cost, InitBal, Natlim, Qtylim

ASSUME /\ V1Cost \in Nat
       /\ InitBal \in Nat
       /\ Natlim \in Nat
       /\ Qtylim \in Nat
       /\ V1Cost > 0
       /\ Natlim > 0
       /\ Qtylim > 0
-----------------------------------------------------------------------------

PNCounter(dom) == [p: [dom -> Nat],
                   n: [dom -> Nat]]

InitPNCounter(dom) == [p |-> [d \in dom |-> 0], n |-> [d \in dom |-> 0] ]


SumAll(map) ==
    LET Sum[r \in SUBSET DOMAIN map] == IF r ={} THEN 0 ELSE LET y == CHOOSE x \in r: TRUE IN map[y]+ Sum[r\
    IN Sum[DOMAIN map]

EvalPNCounter(pnc) == SumAll(pnc.p) - SumAll(pnc.n)

Max(n1,n2) == IF n1>= n2 THEN n1 ELSE n2

MergePNCounters(pnc1, pnc2) == [p |-> [d \in DOMAIN pnc1.p |-> Max(pnc1.p[d], pnc2.p[d])], n|-> [d \in DOMAI

-----------------------------------------------------------------------------

Wallet == [balance: PNCounter(Replicas),
           v1cnt: PNCounter(Replicas),
           vecclc: [Replicas ->Nat] ]


TypeInv == /\ wallets \in [Replicas->Seq(Wallet)]


Init == \*/\ Print ("a", TRUE)
        /\ wallets = [r \in Replicas |-> <<[balance |-> InitPNCounter(Replicas),
                                            v1cnt  |-> InitPNCounter(Replicas),
                                            vecclc |-> [r2 \in Replicas |-> 0] ]>>]


App(elt, s) == <<elt>> \o s

BuyV1(rep, qty) ==
    LET wr == Head(wallets[rep])
        new_bal_n == [wr.balance.n EXCEPT ! [rep] = wr.balance.n[rep] + qty*V1Cost]
        new_v1_cnt_p == [wr.v1cnt.p EXCEPT ! [rep] = wr.v1cnt.p[rep] + qty]
        new_vc == [wr.vecclc EXCEPT ! [rep] = wr.vecclc[rep]+1]
        wr_new == [balance |-> [p |-> wr.balance.p, n |-> new_bal_n ],
                   v1cnt |-> [p |-> new_v1_cnt_p, n |-> wr.v1cnt.n],
                   vecclc |-> new_vc]
    IN /\ wr.balance.n[rep]+qty*V1Cost <= Natlim
       /\ wr.v1cnt.p[rep]+qty <= Natlim
       /\ wr.vecclc[rep]+1 <= Natlim
       /\ EvalPNCounter(wr.balance)+ InitBal >= qty*V1Cost
       /\ wallets' = [wallets EXCEPT ! [rep] = App(wr_new, wallets[rep]) ]
       /\ Print("Buy", TRUE)
```

```
      \* /\ Print(wallets', TRUE)

GetElt(seq, ind) == Head(SubSeq(seq,ind,ind))

Merge(rep1, rep2, ind) ==
    LET wr1 == Head(wallets[rep1])
        wr2 == GetElt(wallets[rep2], ind)
        new_vc == [r \in Replicas |-> Max(wr1.vecclc[r],wr2.vecclc[r])]
        wr1_new ==[balance |-> MergePNCounters(wr1.balance, wr2.balance),
                   v1cnt |-> MergePNCounters(wr1.v1cnt, wr2.v1cnt),
                   vecclc |-> new_vc]
    IN /\ \E r \in Replicas: wr1.vecclc[r] < wr2.vecclc[r]
       /\ wallets' = [wallets EXCEPT ! [rep1] = App(wr1_new, wallets[rep1]) ]
       /\ Print("Merge", TRUE)
       \*/\ Print(wallets', TRUE)

MergeLastStates(rep1, rep2) ==
    LET wr1 == Head(wallets[rep1])
        wr2 == Head(wallets[rep2])
        new_vc == [r \in Replicas |-> Max(wr1.vecclc[r],wr2.vecclc[r])]
        wr1_new ==[balance |-> MergePNCounters(wr1.balance, wr2.balance),
                   v1cnt |-> MergePNCounters(wr1.v1cnt, wr2.v1cnt),
                   vecclc |-> new_vc]
    IN /\ \E r \in Replicas: wr1.vecclc[r] < wr2.vecclc[r]
       /\ wallets' = [wallets EXCEPT ! [rep1] = App(wr1_new, wallets[rep1]) ]
       /\ Print("MergeLastStates", TRUE)
       \*/\ Print(wallets', TRUE)


\* Amount of Money spent should be equal to the number of vouchers bought times the unit cost of the voucher
ConservationOfMoney == \A r \in Replicas: EvalPNCounter(Head(wallets[r]).balance)  + EvalPNCounter(Head(wall

\* Balance in the wallet is always positive --- DOES NOT HOLD
PosBalance == \A rep \in Replicas: InitBal + EvalPNCounter(Head(wallets[rep]).balance) >= 0

\* Fields of P and N fields of PN counters and vector clocks are monotonically nondecreasing in time
Monotonicity == /\ \A r \in Replicas, r2 \in Replicas, i \in Nat: (i>0 /\ i<Len(wallets[r]) ) => (GetElt(wal
                                                                                                /\ GetElt(wa
                                                                                                /\ GetElt(wa
                                                                                                /\ GetElt(wa
                                                                                                /\ GetElt(wa

FinalState(vc) == \A rep \in Replicas: vc[rep] = Natlim
EqualStates(st1, st2) == \A rep \in Replicas: st1.balance.p[rep] = st2.balance.p[rep] /\ st1.balance.n[rep]
\* Eventually all states converge to the same state
Convergence == \A r1 \in Replicas, r2 \in Replicas: FinalState(Head(wallets[r1]).vecclc) /\ FinalState(Head(

Next == \E r1 \in Replicas, r2 \in Replicas, qty \in 1..Qtylim(*, i \in Nat*): (*i <= Len(wallets[r2]) /\ i

Spec == Init /\ [] [Next]_<<wallets>>

THEOREM Spec => TypeInv /\ ConservationOfMoney
=============================================================================
\* Modification History
\* Last modified Wed Sep 24 18:48:05 EEST 2014 by Suha
\* Created Mon Sep 22 12:30:42 EEST 2014 by Suha
```

A.3.1 Wallet use case using Integers

The balance and the counts of the vouchers in the wallets are integers instead of PN-Counters. When two copies of a wallet account are merged the maximum of the balances of the wallets is taken so the balance is generous for the player. When merging the total count of vouchers, the amount of vouchers is added in the two copies. As expected, when it is tested in TLC some money is lost, i.e. the total money spend is more than the money reduced from the balance. The TLA+ representation for this particular example is presented bellow.

```
----------------------------- MODULE walletNat -----------------------------
EXTENDS Integers, TLC, Naturals
VARIABLE wallets
CONSTANTS Replicas, V1Cost, InitBal, Natlim, Qtylim

(*ASSUME /\ V1Cost \in Nat
       /\ InitBal \in Int
       /\ Natlim \in Nat
       /\ Qtylim \in Nat
       /\ V1Cost > 0
       /\ InitBal > 0
       /\ Natlim > 0
       /\ Qtylim > 0*)

-----------------------------------------------------------------------------

Wallet == [balance: Int,
           v1cnt: Int,
           vecclc: [Replicas->Nat]]

TypeInv == /\ wallets \in [Replicas -> Wallet]

Init ==  /\ wallets = [r \in Replicas |-> [balance |-> InitBal,
                                           v1cnt |-> 0,
                                           vecclc |-> [r2 \in Replicas |-> 0] ] ]
        /\Print(wallets, TRUE)

BuyV1(rep, qty) ==
    LET wr == wallets[rep]
        new_vc == [wr.vecclc EXCEPT ! [rep] = wr.vecclc[rep] + 1]
        new_wr == [balance |-> wr.balance - qty*V1Cost,
                   v1cnt |-> wr.v1cnt + qty,
                   vecclc |-> new_vc]
    IN /\ wr.balance >= qty*V1Cost
       /\ wr.v1cnt + qty <= Natlim
       /\ wr.vecclc[rep]+1 <= Natlim
       /\ wallets' = [wallets EXCEPT ! [rep] = new_wr]
      \* /\ Print("Buy", TRUE)
       \*/\ Print(wallets', TRUE)

Max(n1,n2) == IF n1>= n2 THEN n1 ELSE n2

Merge(r1,r2) ==
    LET wr1 == wallets[r1]
        wr2 == wallets[r2]
        new_vc == [r \in Replicas |-> Max(wr1.vecclc[r], wr2.vecclc[r])]
        new_w1 == [balance |-> Max(wr1.balance, wr2.balance),
                   v1cnt |-> Max(wr1.v1cnt, wr2.v1cnt), \* wr1.v1cnt+wr2.v1cnt, \*alternative formulation
                   vecclc |-> new_vc]

    IN /\ \E r \in Replicas: wr1.vecclc[r] < wr2.vecclc[r]
       /\ wallets' = [wallets EXCEPT ! [r1] = new_w1]
       \*/\ Print("Merge", TRUE)
       \*/\ Print(wallets', TRUE)


ConservationOfMoney == \A r \in Replicas: (InitBal - wallets[r].balance) >= wallets[r].v1cnt*V1Cost

FinalState(vc) == \A r \in Replicas: vc[r] = Natlim
EqualStates(st1,st2) == st1.balance = st2.balance /\ st1.v1cnt = st2.v1cnt
Convergence == \A r1 \in Replicas, r2 \in Replicas: FinalState(wallets[r1].vecclc) /\ FinalState(wallets[r2]
Next == \E r1 \in Replicas, r2 \in Replicas, qty \in 1..Qtylim: (BuyV1(r1,qty) \/ Merge(r1,r2))

Spec == Init /\ [] [Next]_<<wallets>>

THEOREM Spec => TypeInv /\ ConservationOfMoney /\ Convergence
=============================================================================
\* Modification History
\* Last modified Thu Sep 25 14:44:20 EEST 2014 by Suha
\* Created Thu Sep 25 13:51:16 EEST 2014 by Suha
```

A.3.2 Wallet use case using No Atomic Operations

In here it is presented the scenario in which buying some vouchers is not an atomic operation. For this model, it is defined two operators so that PN-Counters, which keep the balance and voucher counts, could be merged separately. As expected again, some money is lost and some properties are not satisfied. The TLA+ representation for this particular example is presented bellow.

```
----------------------------- MODULE walletWOTx -----------------------------
EXTENDS Naturals, TLC
VARIABLES wallets
CONSTANTS Replicas, V1Cost, InitBal, Natlim, Qtylim

ASSUME /\ V1Cost \in Nat
       /\ InitBal \in Nat
       /\ Natlim \in Nat
       /\ Qtylim \in Nat
       /\ V1Cost > 0
       /\ Natlim > 0
       /\ Qtylim > 0
-----------------------------------------------------------------------------

PNCounter(dom) == [p: [dom -> Nat],
                   n: [dom -> Nat]]

InitPNCounter(dom) == [p |-> [d \in dom |-> 0], n |-> [d \in dom |-> 0] ]


SumAll(map) ==
    LET Sum[r \in SUBSET DOMAIN map] == IF r ={} THEN 0 ELSE LET y == CHOOSE x \in r: TRUE IN map[y]+ Sum[r\
    IN Sum[DOMAIN map]

EvalPNCounter(pnc) == SumAll(pnc.p) - SumAll(pnc.n)

Max(n1,n2) == IF n1>= n2 THEN n1 ELSE n2

MergePNCounters(pnc1, pnc2) == [p |-> [d \in DOMAIN pnc1.p |-> Max(pnc1.p[d], pnc2.p[d])], n|-> [d \in DOMAI

-----------------------------------------------------------------------------

Wallet == [balance: PNCounter(Replicas),
           v1cnt: PNCounter(Replicas),
           vecclc: [Replicas ->Nat] ]


TypeInv == /\ wallets \in [Replicas->Wallet]


Init == \*/\ Print ("a", TRUE)
        /\ wallets = [r \in Replicas |-> [balance |-> InitPNCounter(Replicas),
                                          v1cnt  |-> InitPNCounter(Replicas),
                                          vecclc |-> [r2 \in Replicas |-> 0] ]]
```

```
BuyV1(rep, qty) ==
    LET wr == wallets[rep]
        new_bal_n == [wr.balance.n EXCEPT ! [rep] = wr.balance.n[rep] + qty*V1Cost]
        new_v1_cnt_p == [wr.v1cnt.p EXCEPT ! [rep] = wr.v1cnt.p[rep] + qty]
        new_vc == [wr.vecclc EXCEPT ! [rep] = wr.vecclc[rep]+1]
        wr_new == [balance |-> [p |-> wr.balance.p, n |-> new_bal_n ],
                   v1cnt |-> [p |-> new_v1_cnt_p, n |-> wr.v1cnt.n],
                   vecclc |-> new_vc]
    IN /\ wr.balance.n[rep]+qty*V1Cost <= Natlim
       /\ wr.v1cnt.p[rep]+qty <= Natlim
       /\ wr.vecclc[rep]+1 <= Natlim
       /\ EvalPNCounter(wr.balance)+ InitBal >= qty*V1Cost
       /\ wallets' = [wallets EXCEPT ! [rep] = wr_new ]
       /\ Print("Buy", TRUE)
      \* /\ Print(wallets', TRUE)

Merge(rep1, rep2) ==
    LET wr1 == wallets[rep1]
        wr2 == wallets[rep2]
        new_vc == [r \in Replicas |-> Max(wr1.vecclc[r],wr2.vecclc[r])]
        wr1_new ==[balance |-> MergePNCounters(wr1.balance, wr2.balance),
                   v1cnt |-> MergePNCounters(wr1.v1cnt, wr2.v1cnt),
                   vecclc |-> new_vc]
    IN /\ \E r \in Replicas: wr1.vecclc[r] < wr2.vecclc[r]
       /\ wallets' = [wallets EXCEPT ! [rep1] = wr1_new]
       /\ Print("MergeLastStates", TRUE)
       \*/\ Print(wallets', TRUE)

MergeBalance(rep1, rep2) ==
    LET wr1 == wallets[rep1]
        wr2 == wallets[rep2]
        new_vc == [r \in Replicas |-> Max(wr1.vecclc[r],wr2.vecclc[r])]
        wr1_new ==[balance |-> MergePNCounters(wr1.balance, wr2.balance),
                   v1cnt |-> wr1.v1cnt,
                   vecclc |-> new_vc]
    IN /\ \E r \in Replicas: wr1.balance.n[r] < wr2.balance.n[r]
       /\ wallets' = [wallets EXCEPT ! [rep1] = wr1_new]
       /\ Print("MergeBalance", TRUE)
       \*/\ Print(wallets', TRUE)

MergeCount(rep1, rep2) ==
    LET wr1 == wallets[rep1]
        wr2 == wallets[rep2]
        new_vc == [r \in Replicas |-> Max(wr1.vecclc[r],wr2.vecclc[r])]
        wr1_new ==[balance |-> wr1.balance,
                   v1cnt |-> MergePNCounters(wr1.v1cnt, wr2.v1cnt),
                   vecclc |-> new_vc]
    IN /\ \E r \in Replicas: wr1.v1cnt.p[r] < wr2.v1cnt.p[r]
       /\ wallets' = [wallets EXCEPT ! [rep1] = wr1_new]
       /\ Print("MergeBalance", TRUE)
       \*/\ Print(wallets', TRUE)



\* Amount of Money spent should be equal to the number of vouchers bought times the unit cost of the voucher
ConservationOfMoney == \A r \in Replicas: EvalPNCounter(wallets[r].balance)  + EvalPNCounter(wallets[r].v1cn

\* Balance in the wallet is always positive --- DOES NOT HOLD
PosBalance == \A rep \in Replicas: InitBal + EvalPNCounter(wallets[rep].balance) >= 0

FinalState(vc) == \A rep \in Replicas: vc[rep] = Natlim
EqualStates(st1, st2) == \A rep \in Replicas: st1.balance.p[rep] = st2.balance.p[rep] /\ st1.balance.n[rep]
\* Eventually all states converge to the same state ---DOES NOT HOLD
```

```
Convergence == \A r1 \in Replicas, r2 \in Replicas: FinalState(wallets[r1].vecclc) /\ FinalState(wallets[r2]

Next == \E r1 \in Replicas, r2 \in Replicas, qty \in 1..Qtylim: (BuyV1(r1,qty) \/ MergeCount(r1,r2) \/ Merge

Spec == Init /\ [] [Next]_<<wallets>>

THEOREM Spec => TypeInv /\ ConservationOfMoney /\ PosBalance /\ Convergence

=============================================================================
\* Modification History
\* Last modified Thu Sep 25 21:36:16 EEST 2014 by Suha
\* Created Thu Sep 25 21:01:10 EEST 2014 by Suha
```

A.4 Shared Medicine Record (FMK)

```
---------------------------- MODULE fmk ----------------------------
EXTENDS Naturals, Sequences
CONSTANTS       DC,             \* Set of all DataCenters
                        Pha,    \* Set of all Pharmacies
                        Pat,    \* Set of all Patients
                        Tre,    \* Set of all Treatments
                        Pre,    \* Set of all Prescriptions
                        Doc,    \* Set of all Doctors
                        MAX     \* maximun clock

VARIABLE patientdb, clock

ASSUME MAX \in Nat


TimeStamps == DC \times (1..MAX)

\* Initialize Variables
Init ==
        /\ patientdb = [d \in DC |-> [ p \in Pat |-> [treat |-> {},
                                                                        presc |-> {},
                                                                        taken |-> {} ] ] ]
        /\ clock = [ d \in DC |-> 0 ]


\* Local Operation at the datacenter d that represents adding a treatment t to patient p by doctor doc
\* Pre:         - doc, p, t all exist and belong to their set.
\* Post:        - The patient p treatment t is updated in the local DC d.
\*                      - The logical clock is incremented by one.

addTreatment(dc, patient, doctor, treatment) ==
        LET
                timestamp == << dc, clock[dc] + 1 >>
                s == patientdb[dc]
                t == << doctor, treatment, timestamp >>
        IN
                /\ patientdb' = [ patientdb EXCEPT ![dc][patient].treat = s[patient].treat \cup {t} ]
                /\ clock' = [clock EXCEPT ![dc] = clock[dc] + 1]



\* Local Operation at the datacenter d that represents adding a prescription pres to the treatment <<doc, t,
\* Pre:         - doc, pres, t all exist and belong to their set.
\* Post:        - If the <<doc, t, date>> exists it adds a the prescription.

addPrescription(dc, patient, doctor, treatment, timestamp, prescription) ==
        LET
                p == << doctor, treatment, timestamp, prescription>>
                s == patientdb[dc]
        IN
            /\ << doctor, treatment, timestamp >> \in s[patient].treat
        /\ p \notin s[patient].presc
        /\ patientdb' = [patientdb EXCEPT ![dc][patient].presc = s[patient].presc \cup {p} ]
                /\ UNCHANGED<<clock>>

\* Local Operation at the datacenter d that represents consuming a prescription pres in a pharmacy f by a pa
\* Pre:         - f, p, pres all exists and belongs to their set.
\* Post:        -

giveDrug(dc, patient, doctor, treatment, timestamp, prescription, pharmacy) ==
        LET
        timestamp2 == << dc, clock[dc] + 1 >>
        p ==  << doctor, treatment, timestamp, prescription, pharmacy, timestamp2 >>
                s == patientdb[dc]
        IN
                /\ << doctor, treatment, timestamp, prescription>> \in s[patient].presc
        /\ \E ts \in TimeStamps : << doctor, treatment, timestamp, prescription, pharmacy, ts >> \in s[patie
        /\ patientdb' = [ patientdb EXCEPT ![dc][patient].taken = s[patient].taken \cup {p} ]
                /\ clock' = [clock EXCEPT ![dc] = clock[dc] + 1]


\* Merge two datacenter databases
merge(dc1, dc2) ==
    LET
        s1 == patientdb[dc1]
        s2 == patientdb[dc2]
        ns == [ p \in Pat |-> [treat |-> s1[p].treat \cup s2[p].treat,
                               presc |-> s1[p].presc \cup s2[p].presc,
                               taken |-> s1[p].taken \cup s2[p].taken ] ]
    IN
        /\ patientdb' = [ patientdb EXCEPT ![dc1] = ns, ![dc2] = ns]
        /\ UNCHANGED<<clock>>

NoDrugGivenTwice == \A d \in DC : \E p \in Pat: \E t1 \in patientdb[d][p].taken, t2 \in patientdb[d][p].take
    t1 /= t2 /\ SubSeq(t1, 1, 5) =  SubSeq(t2, 1, 5)     \*  DO NOT HOLD

Next ==
    \/ \E dc \in DC, patient \in Pat, doctor \in Doc, treatment \in Tre : addTreatment(dc,patient,doctor,tre
    \/ \E dc \in DC, patient \in Pat, doctor \in Doc, treatment \in Tre, ts \in TimeStamps, prescription \in
            addPrescription(dc,patient,doctor,treatment,ts,prescription)
    \/ \E dc \in DC, patient \in Pat, doctor \in Doc, treatment \in Tre, ts \in TimeStamps, prescription \in
            giveDrug(dc,patient,doctor,treatment,ts,prescription,pharmacy)

Spec == Init /\ [][Next]_<<patientdb,clock>>

THEOREM Spec => []NoDrugGivenTwice \*This has to be defined
=============================================================================
```

A.5 Festival

A.6 Business to Business (B2B)