Project no.	609551
Project acronym:	SyncFree
Project title:	*Large-scale computation without synchronisation*

European Seventh Framework Programme

ICT call 10

Deliverable reference number and title:	D.1.2 Formal-language requirements
Due date of deliverable:	April 1, 2015
Actual submission date:	May, 2015
Start date of project:	October 1, 2013
Duration:	36 months
Name and organisation of lead editor for this deliverable:	Koç University
Revision:	0.1
Dissemination level:	PU

Contents

1 Executive Summary

In work package 1 of the SyncFree project, we have gathered functional requirements of a range of large-scale distributed applications. These applications have been implemented recently or are in the process of being implemented by the SyncFree industrial partners. The research, development, and experimentation within the SyncFree project is tightly connected to and guided by a set of industrial use cases. These use cases and natural-language specifications of desired corrrectness and consistency properties were the topic of D1.1. The goal of this document, D1.2, is to take the next refinement steps and to express these specifications first in a combination of ordinary mathematics and natural language, and then fully formally using a temporal logic.

The natural-language requirements for the applications in our use cases had inevitably been influenced by existing or ongoing implementations. These implementations make use of distributed data types and databases and aim to prioritize availability over strong consistency, but their design and implementation has not been carried out with conflict-free data types (CRDTs) and the desirable guarantees that CRDTs provide in mind. In the work described in this document, we first revisit each use case with CRDTs in mind, and make more precise and formal the requirements for each application using an intuitive combination of natural language and mathematics. These requirements are of the form of replica-local and global invariants, where the latter pertain to the converged final state of the system. We then make the data type, system and application models, and desired properties completely formal, using the Temporal Logic of Actions (TLA+, [3]) as the formal modeling and property specification language.

These semi-formal and formal specifications serve as guides for the verification and system-building activity in SyncFree. The semi-formal specifications are more useful when considering design and implementation trade-offs, e.g. when placing bounds on divergence. The fully formal TLA+ specifications serve a dual purpose. First, they are unambiguous, executable operational specifications for the use cases. Second, they have allowed the use of formal verification tools such as the TLC model checker on an abstract model of the system to verify application invariants and to detect violations of such invariants.

2 Milestones in the Deliverable

WP1 has reached the following milestones:

Mil. no	**Milestone name**	**WP**	**Date due**	**Actual date**
S1	CRDT consolidation in a static environment	WP2	M12	M12

The corresponding tasks are:

Task no	**Task name**	**Date due**	**Actual date**	**Leader**
D.1.1	Natural-language requirements	M06	M06	Trifork
D.1.2	Formal-language requirements	M18	M18	Trifork

3 Contractors Contributing to the Deliverable

The following people contributed to this deliverables: Peter Zeller (KL)
Carla Ferreira (Nova)
Maryam Dabaghchian, Erdal Mutlu, Suha Orhun Mutluergil, Burcu Ozkan, Serdar Tasiran (KU)
Jordi Martori (Inria)
Tom Benedictus, Amadeo Ascó (Trifork)

4 Introduction and Preliminaries

In this document, for the industrial use cases in SyncFree, the informal requirements in D1.1 are refined into

- semi-formal requirements expressed in a combination of natural language descriptions and mathematical inequalities, and
- (for four of the six use cases) fully-formal requirements expressed in the Temporal Logic of Actions.

The semi-formal requirements for existing applications and applications currently in development have been documented anticipating the use of CRDTs. However, as they stand, these applications are not written using CRDTs exclusively, and they make use of geo-replicated databases in other ways also. We have therefore chosen to write the semi-formal requirements to correspond to the functional correctness requirements of the applications as they stand today. When writing fully formal requirements, we made the choice to make full use of CRDTs and modeled applications, data structures and the correctness requirements using CRDTs as the only data representation. We have modeled state-based CRDTs and formulated requirements as necessary in both of these settings. As part of work package 4, the TLA+ models for four of the use cases have been model checked for small configurations (small number of CRDTs, replicas and operations) using the TLC model checker.

The following constants and variables have been used in the semi-formal descriptions for all use cases.

- DC: The set of all Data Centres (DCs). d identifies one of the DCs, $d \in DC$, where $|DC|$ is the number of DCs.
- DV: The set of devices. dv represents one of those devices, $dv \in DV$, where $|DV|$ is the number of devices.
- $Nodes$: The set of nodes. n represents one of those nodes, $n \in Node$, where $|Nodes|$ is the number of nodes in a DC.
- $Clients$: The set of clients and c represents a client in the system, $c \in Clients$, where $|Clients|$ is the total number of clients.

The industrial partners Rovio and Trifork have both contributed three use cases each to the SyncFree project. These use cases were chosen in order to highlight different significant issues and difficulties when targeting extreme-scale sharing and availability while, at the same time, trying to ensure application correctness. Sections 5-10 present the semi-formal requirements for the use cases. The Appendix contains the fully-formal TLA+ specifications.

5 Advertisement Counter

5.1 Overview

Online advertising platforms need to accurately record the number of times an ad is displayed (number of "impressions") and clicked on in order to analyse advertising data. Such

platforms typically use distributed counters, which are challenging to implement in a dynamic, error-prone environment. Conflict-free Replicated Data Type (CRDT) counters are a promising solution; the challenge is to scale to an extreme numbers of users while ensuring correct operation.

Rovio's Ads service keeps track of impressions and clicks for ads per campaign, per ad, and per 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 while avoiding synchronization.

The main use of the data tracked in this use case 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 early in the campaign. To be able to control the rate at which ads are displayed, each measure kept track of by the system must remain reasonably close to its actual real-time value.

5.2 Requirements

The system is represented by the constants and variables defined in Table 1.

Name	Description	Domain
AD	The set of all advertisement-campaigns. a identifies one of the ads, $a \in AD$, where $\|AD\|$ is the number of ads.	$\mathbb{Z}^+$

Name/Description	Domain
$maxTotalViews(a)$	$\mathbb{Z}^+$
The maximum total number of times the ad a should be shown.	
$maxTotalViewsPerDC(a, d)$	$\mathbb{Z}^+$
The total number of times the ad a should be shown by DC d.	
$maxViewsPerDevice(a)$	$\mathbb{Z}^+$
Represents the maximum number of times the ad a should be presented on a device.	
$viewsPerDevice(a, dv)$	$\mathbb{Z}^+$
The number of times the ad a has been shown on the device g.	
$verifiedViews(a, n, q)$	$\mathbb{Z}^+$
The verified number of times an ad a has been shown by node q as the node n report it, $n, q \in \{1, \ldots, \|Nodes\|\}$.	
$averageViews(a)$	$\mathbb{Z}^+$
The average number of times ad a is shown. The workload is equally spread between all the nodes, $\frac{averageViews(a)}{\|Nodes\|}$.	

Table 1: Ad Counters Constants and Variables.

Equality 1 states that the total maximum number of times an ad should be presented in each data center (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 minimize Δ_a, the difference between the actual number of times ad a has been viewed, and the desired upper bound $maxTotalViews(a)$ on this quantity.

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

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

$$maxTotalViews(a) = \sum_{dv \in DV} viewsPerDevice(a, dv) + \Delta_a \tag{3}$$

Each data center comprises of a set of nodes and the current implementation of the ads service runs on multiple such service nodes. In order to avoid write conflicts between nodes, in the current implementation, each of those nodes keeps a local "document" for the impression and click counters in Riak. This can be viewed as a rudimentary implementation of the counter CRDT. The value of the counter at the data center can be obtained by adding the values of the counters stored in the documents at each service node.

Client applications running on different kinds of devices (e.g., mobile phones, tablets.) connect to the ads service in order to determine which ads to display to their users. The application has the requirement that the same ad should not be shown on the same mobile device more than 3 times in any given day ($maxViewsPerDevice(a, dv) = 3$). To ensure this, for each device and ad, the ads service currently keeps track of the number of times the ad has been shown on that device. Currently, the system keeps track of this information approximately by storing a single document per device in which the time instances each ad has been shown on the device have been recorded. It might be possible to utilise CRDTs to store this information.

The current implementation of the ads system is not particularly elegant or easy to maintain. The CRDT counters are expected to provide a more elegant and efficient solution for updating the counters. The ad system does not really need to enforce a strict limit for the counter and an "optimistic", more available solution can be implemented instead. In this solution, if the *local* view of the counter value is less than the limit, then the ad is shown and the local counter increased. Thus, synchronization to obtain the precise value of the counter is avoided. Since a replica may not have yet been notified of updates at other replicas, this approach may result in showing the ad too many times, but this is an acceptable trade-off in order to accomplish more availability and responsiveness.

In a DC, each server node has its own cache for the counters. The counters get synchronized among nodes via Riak. 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 synchronized on every update they get from the ad servers. The value of each counter at each database may be expressed as a sum of several (e.g. 3) actual counter variables, which are aggregated to yield the actual value of the counter. 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. The current implementation makes use of only one DC is

currently used. 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 last time a synchronization between nodes had been performed. This quantity is a valid lower bound for the current number of times ad a has been shown by node n.

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

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 of times ad a has been seen at any time, $views(a, n)$, as seen by a node n may be less than or equal to the real total number of times the ad a has been shown, $views(a)$, as shown by Inequality 6. When synchronization is performed or the system converges to a quiescent state, this inequality should become an equality.

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

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

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.

$$views(a) = views(a, n) = views(a, m) \quad \forall\ n, m \in \{1, \ldots, |Nodes|\} \tag{7}$$

5.3 Multiple Data Centres

In this section, we present the semi-formal requirements for the case where multiple data centers are employed. The campaign counter could be split between the different DCss in different countries, which will improve the accuracy of the counter (Figure 1). A portion of the overall counter would be assigned to each DC, which will only be able to display specific ad this number of times. The updates to the ad counters are replicated in the other DCss at different intervals.

The distribution of the overall limit on the number of times the ad is to be displayed to the individual data centers could be based on different application-specific criteria, e.g. population size covered by each DC, existing statistics from previous campaigns or studies, costumer preferences, the desire to 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 moves to the coverage area of another DC, its counter is not known by the new servicing DC, potentially showing those ads already seen again. Avoiding this may be costly, requiring each device counter to be replicated between several (not necessarily all) DCss.

The extra variables required for modeling the setup with multiple DCs are presented in Table 2.

The model needs to be extended by the following expressions:

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

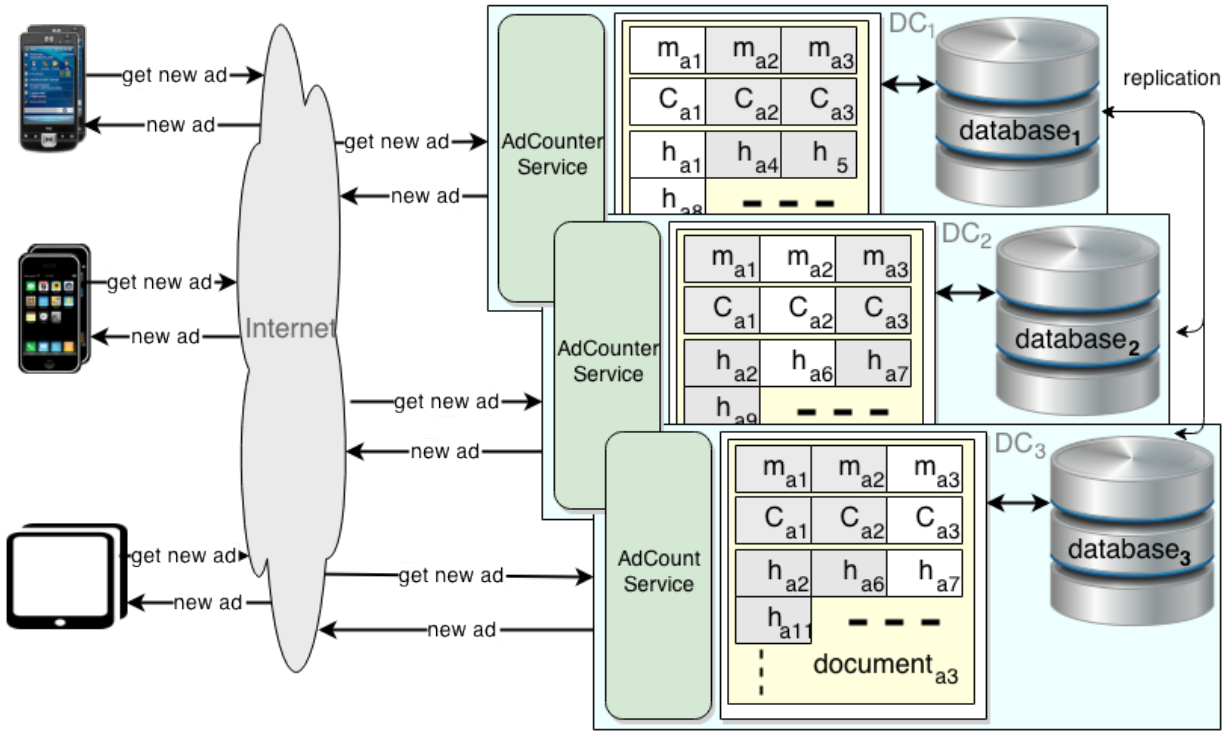

Figure 1: Overview for the distribution of counters with three DCs, $|DC| = 3$, $m_{ad} \equiv targetMaxViews(a, d)$, $C_{ad} \equiv maxViewsPerDevice(a)$ and $h(a, g) \equiv views(a, d, g)$.

$$targetMaxViews(a, d) = views(a, d) \tag{9}$$

$$totalMaxView(a) = \sum_{d \in DC} targetMaxViews(a, d) \tag{10}$$

$$totalViews(a) = \sum_{d \in DC} views(a, d) \tag{11}$$

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. Once the campaign is over, the total number of times an ad has been shown by each DC is the same as the maximum allowed as expressed by Equation 9. Equation 10 states that the ads are distributed through out all the DCs. Equation 11 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.

$$maxTotalViews(a) \geq \sum_{d \in DC} totalViews(a, d) \tag{12}$$

$$maxTotalViews(a) = totalViews(a) + \Delta_a \tag{13}$$

Name/Description	Domain
$Nodes_d$	$\mathbb{Z}^+$
The set of nodes in DC d. n represents one of those nodes, $n \in \{1, \ldots, \lvert Nodes_d \rvert\}$.	
$averageViews(a, d)$	$\mathbb{Z}^+$
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, $\frac{averageViews(a,d)}{\lvert Nodes_d \rvert}$.	
$verifiedViews(a, d, n)$	$\mathbb{Z}^+$
The verified number of times an ad a has been shown by node n in DC d reported by node n as of the last synchronization.	
$targerMaxViews(a, d)$	$\mathbb{Z}^+$
The maximum number of times 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 $targerMaxViews(a, d) > 0$.	
$views(a, d)$	$\mathbb{N}_0$
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.	
$totalViews(a)$	$\mathbb{Z}^+$
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 2: Ad Counters extra Constants and Variable.

Inequality 12 gives the total limit for all the $totalViews(a, d)$ which can be obtained from Inequality 8 and Equation 10, whereas Equation 13 shows that the total number of times ad a has been shown, once the campaign has concluded is equal to the total number ad a should has been shown.

Equation 4 can be generalised for many DCs as shown in Equation 14.

$$views(a, d) = \sum_{n \in Nodes_d} verifiedViews(a, d, n) + averageViews(a, d) \tag{14}$$

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 take into account the state of the data in each of the DCs the definitions from the single data center case are extended and some new symbols are introduced below, Table 3. The discrepancy of the counter for ad a shown on devices by DC d is zero when it is reported by the same DC as expressed in Equation 16.

$$\Delta views(a, d, q) = views(a, d, d) - views(a, d, q) \tag{15}$$

$$\Delta views(a, d, d) = 0 \tag{16}$$

Name/Description	Domain
$views(a, d, q)$	$\mathbb{Z}^+$
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, $d, q \in \{1, \ldots, \lvert DC \rvert\}$.	
$\Delta views(a, d, q)$	$\mathbb{Z}^+$
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 15.	
$\Delta totalViewsDiscrepancy(a, d, q)$	$\mathbb{Z}^+$
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 17.	

Table 3: Ad Counters Constants and Variable for discrepancies in values between DCs.

$$\Delta totalViewsDiscrepancy(a, d) = \sum_{q \in DC} \mid \Delta views(a, d, q) \mid \tag{17}$$

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 14.

$$\Delta totalViewsDiscrepancy(a, d) = 0 \tag{18}$$

6 Leader Board

6.1 Overview

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 for 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.)

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.

6.2 Requirements

A mathematical representation of this use case is represented in Table 4.

Name	Description	Domain
$Games$	The set of games.vg represents a game in the overall system, $g \in Games$, where $\|Games\|$ is the number of games.	$\mathbb{Z}^+$
$Levels_g$	The set of levels in game g. l identifies a level in game g, $l \in Levels_g$, where $\|levels_g\|$ is the number of levels in game g.	$\mathbb{Z}^+$
$Players_{gld}$	The set of players which have played at some stage the game g at level l, as it is seen by DC d. 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.	$\mathbb{Z}^+$
$Score_{gldp}$	The score player p has achieved at level l of game p as seen by DC d.	$\mathbb{Z}^+$
$highestScore_{gld}$	The highest score achieved by all players that have played game g at level l as it is seen by DC d. Calculation of this quantity is shown in Equation 19.	$\mathbb{Z}^+$
$PlayersRank_{gld}$	Represents the group of all the players whose scores for the game g 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. This quantity is made more precise in Equation 20.	$(\mathbb{Z}^n)^+$

Table 4: Leader Board Constants and Variables.

$$highestScore_{gld} = \max\{Score_{gld1}, \ldots, Score_{gld|Players_{gld}|}\} \; \forall g \in Games, l \in Levels_g, d \in DC \tag{19}$$

$$PlayersRank_{gld} = \{j | Score_{gldj} \geq Score_{gldq} \; \forall q \in Players_{gld}\} \tag{20}$$

Furthermore $PlayersRank_{gld}$ could be extended to represent the different positions in the Leader Board, such that $PlayersRank^i_{gld}$ is the group of players which are at position i on the Leader Board, $i \in \mathbb{Z}_{\geq 1}$, as shown in Equation 21. This means that $PlayersRank^1_{gld}$, which represent the top position, is equivalent to $PlayersRank_{gld}$, such that $PlayersRank^1_{gld} = PlayersRank_{gld}$.

$$PlayersRank^i_{gld} = \{j | Score_{gldq} < Score_{gldj} < Score_{gldo} \; \forall q \in PlayersRank^{g-1}_{gld}, o \in PlayersRank^{g+1}_{gld}\}, \; g \in \mathbb{N}_{>1} \tag{21}$$

Equation 22 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.

$$highestScore_{gld} = Score_{gldp} \; \forall \; g \in Games, l \in Levels_g, d \in DC, p \in PlayersRank^1_{gld} \quad (22)$$

7 Virtual Wallet

7.1 Overview

Virtual wallet applications manage virtual economies. Such applications require massive scalability and very robust security guarantees. To ensure very low per-transaction financial cost, as required for use at fine granularity, some consistency constraints may need to be temporarily relaxed. Replicas of a particular wallet each keep a local copy of their state and possibly other replica states, and perform credits and debits based on the information locally available to them. But, in an eventually consistent setting, replica states may not be up to date. The challenge is to maintain correctness at an extreme scale, i.e., to ensure that money does not vanish, or is not created out of thin air, despite data fragmented across numerous replicas, lost or duplicated information, long-term disconnection.

Rovio's wallet service provides a delivery mechanism for in-game items and manages the user's virtual currencies. A 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 (recent) transactions performed on the wallet.

- The balances stored in a wallet each consist of a numerical value and currency name (e.g Crystals: 150 or Euro: 2.45).
- Wallets may also contain vouchers. A voucher consists of a unique voucher ID and item details such as the item name and type. Vouchers are removed from the wallet when consumed.
- A transaction consists of a unique transaction ID, timestamp, transaction type and the additional data is needed for the particular transaction type. A transaction may comprise of the purchase or return of a voucher, a debit or credit for a particular kind of currency. Records of transactions are periodically archived to keep storage requirements reasonable. Records of transactions are only removed from wallets when archived.

The quantities kept track of in a virtual wallet have actual monetary value. Therefore, it is one of the requirements of the virtual wallet system to not result in the loss or creation of money. This requires custom conflict resolution and, most likely, additional mechanisms such as reservations and escrows as investigated in other work packages in SyncFree [1]. Using CRDTss, the balances for each currency can be represented as a map from currency name to value counters represented as CRDTs, and the vouchers and transactions as sets as

presented in [5]. A more precise modeling of a virtual wallet using state and operation-based CRDTs is carried out in the fully formal TLA+ specification for the virtual wallet use case.

The Wallet service supports the following operations:

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

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

Potential conflicts. We have identified the following potential conflicts:

- 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). In this case, a reasonable conflict resolution scheme could be to provide the extra items for free to the user.
- 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.

Atomic update requirements. The transaction log needs to contain entries for all operations. 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 for each operation. 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.

7.2 Requirements

A mathematical representation of this use case is represented below. We present the variables involved in mathematically formulating the requirements and describing the effects of each operation and transaction performed. The operational semantics of the data types and the state transition relations for each operation are made fully precise in the TLA+ description for the virtual wallet.

1. $Balance = \{\langle Crsytals, n1\rangle, \langle Euro, n2\rangle\}$ 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 $b'[curr] = b[curr] - amount$, $b'[cr] = b[cr] \iff cr! = curr$. 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 $b, b' \in B$.

2. 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.

3. 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.

4. 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.

5. 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 23 shows that the net amounts of all currencies should be non-negative.

$$\langle curr, n\rangle \in B'_{ci} = B_{ci} \ominus \widetilde{B}_{ci} \implies n \geq 0\ \forall\ curr \in Curr,\ c \in Clients,\ i \in \{1, \ldots, |DC|\} \tag{23}$$

6. 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 24, where $\setminus$ is the standard multiset difference operator.

$$V'_{ci} = V_{ci} \setminus \widetilde{V}_{ci} \tag{24}$$

7. The consumed set of vouchers should be a subset of gained vouchers of a client c, as shown in Equation 25.

$$\widetilde{V}_{ci} \subseteq V_{ci} \tag{25}$$

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

   For purchase item operation that purchases voucher v:

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

As shown in Equation 26, 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.

$$B_{ci}[curr] > cost \implies \langle B_{ci}, \widetilde{B}_{ci}, V_{ci}, \widetilde{V}_{ci}, T\rangle \xrightarrow{o} \langle B_{ci}, \widetilde{B}_{ci} \oplus \{cost, curr\}, V_{ci} \cup \{v\}, \widetilde{V}_{ci}, T'\rangle \\ where\ v = \langle id, \langle cost, curr\rangle, spec\rangle \in V. \quad (26)$$

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 27, 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.

$$\langle B_{ci}, \widetilde{B}_{ci}, V_{ci}, \widetilde{V}_{ci}, T\rangle \xrightarrow{o} \langle B_{ci} \oplus \{amount, curr\}, \widetilde{B}_{ci}, V_{ci}, \widetilde{V}_{ci}, T'\rangle \quad (27)$$

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 28, 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.

$$v \in V_{ci} \implies \langle B_{ci}, \widetilde{B}_{ci}, V_{ci}, \widetilde{V}_{ci}, T\rangle \xrightarrow{o} \langle B_{ci}, \widetilde{B}_{ci}, V_{ci}, \widetilde{V}_{ci} \cup \{v\}, T'\rangle \quad (28)$$

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

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

As shown in Equation 29, 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.

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

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.

8 Shared Medical Records (FMK)

8.1 Overview

The shared medical records use case one of three industrial case studies provided by Trifork. As is the case with the other Trifork use cases, shared medical records (FMK) presents a unique set of requirements and challenges for applications written using CRDTs.

For each person, FMK maintains 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. Not all treatments require prescriptions. But all advice given and medication prescribed by a doctor do a patient will be recorded in the FMK system. "Events" as used in the FMK context are actual events that have taken place 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 FMK system is not a narrow purpose electronic health record system for only storing specialized information such as test results, measurements and the like.

One of the primary design criteria for FMK is to provide high availability. The system is in continuous, non-stop use by a large number of clients, and is currently integrated with more than 40 other healthcare systems, most of which are required to use FMK as the primary storage for relevant medical data.

Although the system is simple when viewed at a 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 Eventual Consistency (EC) with 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 and dataset in each major geographical region and hospital, which still remains an eventual goal. Writes should be propagated as soon as possible, but lack of such propagation (e.g. due to 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 [6, 2], which increases the relevance of the FMK in the support of the national healthcare services and the empowering of patients.

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.

Potential Conflicts 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. Such conflicts are required to be resolved by a doctor making the decision on the final state of the prescription.

Name	Description	Domain
$Patients_d$	The set of patients in FMK as seen by DC d. 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.	$\mathbb{Z}^+$
$Treatments_{dp}$	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.	
$Prescriptions_{dpt}$	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.	
$Prescriptions_{dptr}$	The prescription r in treatment t for patient p as seen by DC d ($Prescriptions_{dptr} \in Prescriptions_{dpt}$ and $r \in \{1, \ldots, \|Prescriptions_{dpt}\|\}$).	$\mathbb{Z}^+$
$Events_{dpt}$	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.	
$Events_{dpte}$	The prescription e for treatment t of patient p as seen by DC d ($Events_{dpte} \in Events_{dpt}$ and $e \in \{1, \ldots, \|Events_{dpt}\|\}$).	$\mathbb{Z}^+$
$Treatments_{dpt}$	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.	$\mathbb{Z}^+$

Table 5: FMK Constants and Variables.

8.2 Requirements

The variables used in formalizing the requirements for the FMK use case are presented in Table 5.

The following is a partial list of operations that can be carried out using the FMK system.

- **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 30. The patient record must already exist in the FMK system.

$$Treatments_{dp} = Treatments_{dp} \cup Treatments_{dpt} \text{ such that } d \in \{1, \ldots, |DC|\}, p \in \{1, \ldots, |Patients_d|\}, t \notin Treatments_{dp} \quad (30)$$

- **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 31. Also the patient and treatment records must already exist on the FMK system.

$$Prescriptions_{dpt} = Prescriptions_{dpt} \cup Prescription_{dptr} \text{ such that } d \in \{1, \ldots, |DC|\}, p \in \{1, \ldots, |Patients_d|\}, t \in \{1, \ldots, |Treatments_{dp}|\}, r \notin Prescription_{dpt} \quad (31)$$

- **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 32. The patient and treatment must already exist in the FMK system.

$$Events_{dpt} = Events_{dpt} \cup Events_{dpte} \text{ such that } d \in \{1, \ldots, |DC|\}, p \in \{1, \ldots, |Patients_d|\}, t \in \{1, \ldots, |Treatments_{dp}|\} e \notin Events_{dpt} \quad (32)$$

Given that prescriptions and events correspond to things events that have already happened, they cannot be removed from a patient treatment, and the system consqeuently has 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:

- Different treatment(s) in either or both of the records seen by any two DCs d and q. Equation 33 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.

$$\exists\, t \in 1, \ldots, |Treatments_{dp}|,\ Treatments_{dpt} \notin Treatments_{qp} \quad (33)$$

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

– Difference(s) between the prescriptions within a treatment t for a patient p. Equation 34 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.

$$\exists\ r \in 1, \ldots, |Prescriptions_{dp}|,\ Prescriptions_{dptr} \notin Prescriptions_{qpt} \tag{34}$$

– Difference(s) between the events within a treatment t for a patient p. Equation 35 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.

$$\exists\ e \in 1, \ldots, |Events_{dp}|,\ Events_{dpte} \notin Events_{qpt} \tag{35}$$

So a patient record is out of sync if any combination of the cases above appear for a patient between different DCs. This model for the FMK system and the set of requirements above have been fully formalized in the TLA+ model.

9 The Festival Use Case

9.1 Overview

The Trifork Festival application is an existing app developed for the Android and iOS platforms (Figure 2). The app 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 and 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 a mobile client is not able to distinguish if he receives a particular vote more than once, since the identity of the voter is not transmitted along with the vote and the network may not be reliable and result in redundant re-transmissions.

Massive events like conferences, sport events, and music festival 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 Bluetooth and WiFi Direct with other devices in a peer-to-peer fashion. So although a mobile client may not be able to connect to a DC, the client can still post his votes to peer devices just as he can get more up to date data from them.

Conventional counters for keeping tallies of votes is not a suitable approach, since we don't have a central database and therefore no means to check how many times a particular client votes. Nor do we have any means to see how large a percentage of possible votes have been cast. In this setting, the use of a probabilistic counter may make more sense.

In the envisioned scenario, 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 devices' bit arrays as a simple bitwise and operation. At any device you can now see the total number of

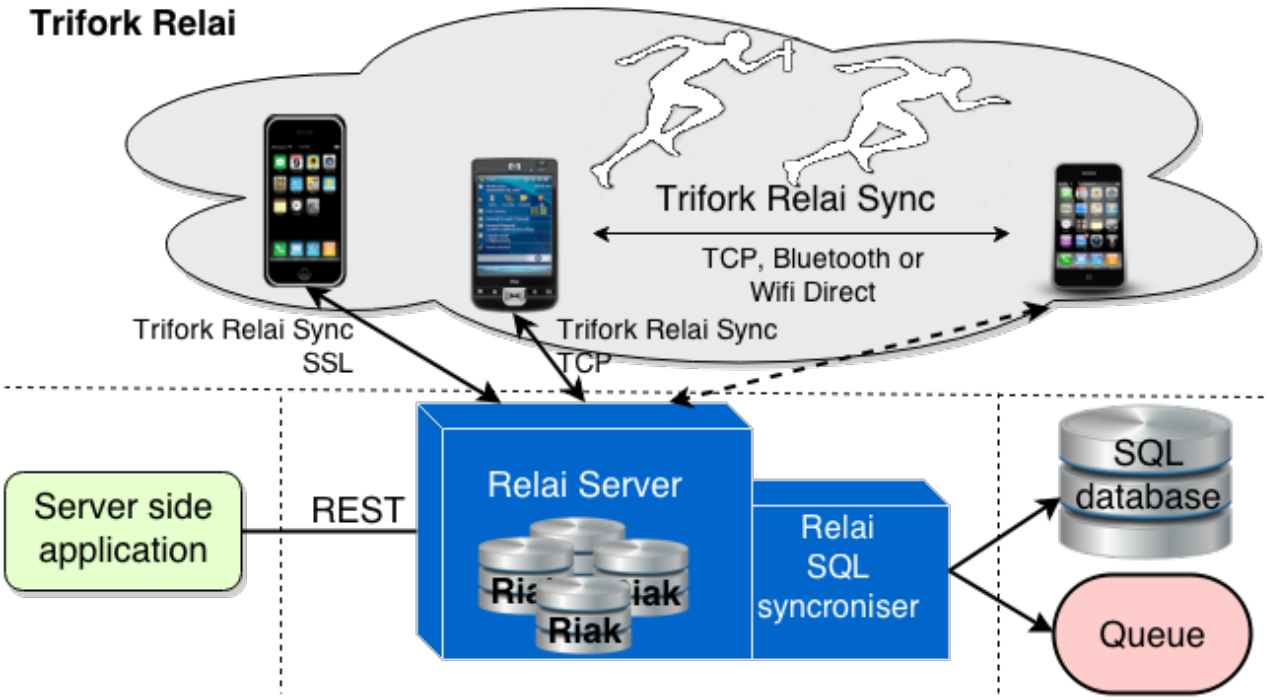

Figure 2: Overview of current Festival implementation which uses Trifork's Relai.

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, the operation for registering a vote is idempotent.

One consistency issue that is left unresolved in this approach is that one cannot determine whether 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. For instance, suppose that no cellular radio network is available and one group are now only networking using WiFi Direct and another group are only networking using Bluetooth and no device is bridging the two means of networking. Then each group will have their own voting polls.

There are a number of unintended potential side effects. For a music festival this is acceptable, while for an app for voting in a parliament this would not be acceptable. In the implementation each device can only vote for each concert once and one cannot alter one's vote after it has been cast. If one has more devices, he also can cast more votes. If one uninstalls and reinstalls the app, one will be able to vote again for the same concert.

9.2 Mark-Counters

A Mark-Counter B is composed of an array of n bits each represented by $B_i \in \{0,1\} \forall i \in \{1,\ldots,n\}$. 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.

The bitwise OR operator is represented as | in this document.
The operations provided by a mark counter are listed below.

- **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.

- **Commutativity**. 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-Counters by B_i and C_i, respectively, 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 36.

$$B_i|C_i = C_i|B_i \tag{36}$$

  The commutativity of the merge operation follows directly from the commutativity of the or operation for each bit.
- **Associativity**. Equation 37 below clearly indicates that since the i-th bit of the result of merging three Mark-Counters is the logical or of the ith bits of each Mark-Counter, the merge operation is associative.

$$(B_i|C_i)|D_i = B_i|(C_i|D_i) \tag{37}$$

- **Idempotence**. The idempotence of the merge operation follows from equation 38 describing the operation on the ith bit when a Mark-Counter is merged with itself.

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

9.3 Requirements

Consider a festival composed of many acts and events. To simplify presentation, we first consider the case of a festival consisting of a single event in a theatre and the constants and variables used are presented in Table 6.

$$\delta_d^{good} = \begin{cases} 1 & if \sum_{a \in People} G_{da} > 0 \\ 0 & otherwise \end{cases} \tag{41}$$

Name	Description	Domain
$People$	The set of people attending the festival. p represents one attendee ($p \in People$), with $\|People\|$ representing the number of attendees and the size of the array of bits.	$\mathbb{Z}^+$
n_d	The highest attendee's index to the festival that the device d has information about, $d \in \{1, \ldots, \|People\|\}$. Also $1 \leq n_d \leq \|People\| \ \forall \ d \in \{1, \ldots, \|People\|\}$.	$\mathbb{Z}^+$
G_d	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, to state the vote of the attendee as good.	$(\mathbb{Z}^+)^{\geq n_d}$
G_{da}	The value in the array of good votes available at devise d for attendee a, presented in Expression 39. Alternatively, the bit a in the array of bits G_d, $a \in \{1, \ldots, \|People\|\}$. $$0 \leq G_{ka} \leq 1 \ \forall k, a \in \{1, \ldots, \|People\|\} \quad (39)$$	$\mathbb{Z}^+$
B_d	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 represents a device. B_d is a Mark-Counter where a bit is set to 1 if the corresponding attendee, represented by that bit, to state the vote of the attendee as bad.	$(\mathbb{Z}^+)^{\geq n_d}$
B_{da}	The value in the array of bad votes seen by device d for attendee a, B_{da}, presented in Expression 40. Also it can be said to be the bit a in the array of bits B_d, $a \in \{1, \ldots, \|People\|\}$. $$0 \leq B_{da} \leq 1 \ \forall d, a \in \{1, \ldots, \|People\|\} \quad (40)$$	$\mathbb{Z}_+$
$numGood_d$	The number of good votes received by the attendee d device, as shown by Equation 43.	$\mathbb{Z}^+$
$numGood$	The total number of attendees that voted good, as shown in Equations 41 and 42. If $n_d < \|People\|$ then $G_{da} = B_{da} = 0 \ \forall \ a > n_d$.	$\mathbb{Z}^+$
$numBad_d$	The number of bad votes received by the attendee d device, as explained in Equation 46.	$\mathbb{Z}_+$
$numBad$	The total number of attendees that voted bad, as shown in Equations 44 and 45.	$\mathbb{Z}^+$
$numAttendees_d$	The total number of attendees that voted as seen by device/attendee d, as shown in Equation 47.	$\mathbb{Z}^+$
$numAttendees$	The total number of attendees that voted, as shown in Equation 48.	$\mathbb{Z}^+$

Table 6: Festival Constants and Variables.

$$numGood = \sum_{d \in People} \delta_d^{good} \tag{42}$$

$$numGood_d = \sum_{a=1}^{n_d} G_{da} \ \forall \ d \in \{1, \ldots, |People|\} \tag{43}$$

$$\delta_d^{bad} = \begin{cases} 1 & if \sum_{d \in People} B_{da} > 0 \\ 0 & otherwise \end{cases} \tag{44}$$

$$numBad = \sum_{d \in People} \delta_d^{bad} \tag{45}$$

$$numBad_d = \sum_{a=1}^{n_d} B_{da} \ \forall \ d \in \{1, \ldots, |People|\} \tag{46}$$

$$numAttendees_d = \sum_{a=1}^{n_d} (G_{da} + B_{da}) \ \forall \ d \in \{1, \ldots, |People|\} \tag{47}$$

$$numAttendees = numGood + numBad \tag{48}$$

An attendee may only vote once as expressed in Inequality 49, but he/she may not vote at all.

$$G_{da} + B_{da} < 2 \ \forall \ d \in \{1, \ldots, |People|\}, a \in \{1, \ldots, n_k\} \tag{49}$$

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 50. All the devices are in sync if $\triangle numAttendees_{dj} = 0 \ \forall \ d, j \in \{1, \ldots, |People|\}$.

$$\triangle numAttendees_{dj} = \sum_{i \in People} ((G_{di} + G_{ji})\%2 + (B_{dj} + B_{ji})\%2) \forall \ d, j \in \{1, \ldots, |People|\} \tag{50}$$

This formalization can be extended to handle multiple events by introducing a new index that represents each of these events.

One issue with the current festival use case is that not all devices may know about all the attendees, and even if they did, the bit arrays required may be too large to be practical. Trifork has therefore devised the Statistical Mark-Counter data structure, briefly presented next. In Trifork's festival the counters are reduced in size by using statistics, and each bit is randomly assigned to a customer.

To reduce the amount of data transferred between devices and for cases where not all devices may know the total number of attendees, a probabilistic approach can be used. Trifork's current festival implementation already uses a 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 to the same position in the poll, so potentially their votes would equate to a single vote, if their votes

are the same. Given this, inequality 51 would not be complied with so it would need to be removed from the representation to 44.

$$G_{da} + B_{da} \leq 2 \ \forall \ d, a \in \{1, \ldots, n\} \tag{51}$$

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.

10 A Business to Business (B2B) Use Case

10.1 Overview

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

The Business to Business (B2B) order application enables the clients, e.g. shop employees, to see a catalogue of upcoming clothes on a tablet device, and place orders for future delivery. 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.

10.2 Formalization

The variables used in the formalization of this use case are shown in Table 7.

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

Table 7: B2B Constants and Variables.

Constraints: It is not allowed to order more boxes for a particular kind of clothing item c from a business b than is available as expressed in inequality 52.

$$Caplcity_{dbl} \geq \sum_{c \in Clients_{db}} Boxes_{dblc} \tag{52}$$

No client is allowed to increase their orders orders more that the credit they have available from business b as expressed by Inequality 53.

$$Credit_{dbc} \geq \sum_{l \in Clothes_{db}} (Boxes_{dblc} * Cost_{dbl}) \tag{53}$$

Orders are modelled as a State-based increment-only Counter (G-set) [4] CRDT which is updated by adding event objects.

The tablets can operate off-line and only need to be online for receiving, for instance, catalogue updates and for posting orders to 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 duplicate orders, items being out of stock when the order is received by the server, catalogues being out of date, and other similar conflicts. The automated system is not intended for handling these conflicts while it will attempt to detect conflicts and other 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.

11 Conclusion

We have described using a combination of mathematical notation and natural language the functional and consistency requirements for six industrial use cases. These use cases have made use of databases residing in one DC, in replicated DCs, on mobile phones, and distributed across all with potential eventual consistency. These use cases also illustrate how data present at a client can possibly believed by the user to be "centrally" persisted whereas it may never be delivered. These scenarios cover the use of CRDTs on the server side, points-of-presence, and at the edge of the network.

Some of the requirements we elicited will be supported and verified early on in SyncFree. Others are more involved and will only be verified later in the project.

All of the examples used as input to the requirement gathering are from the real world and come from global businesses with large presence and market share in the entertainment industry and the health care sector. The current IT Industry as it stands has very little support for the kind of applications examined in this document, and, as we have seen, the problems addressed in the use cases are not supported in classic database implementations.

Fully formal specifications for four of the use cases studied and the platforms they run on are presented in the Appendix.

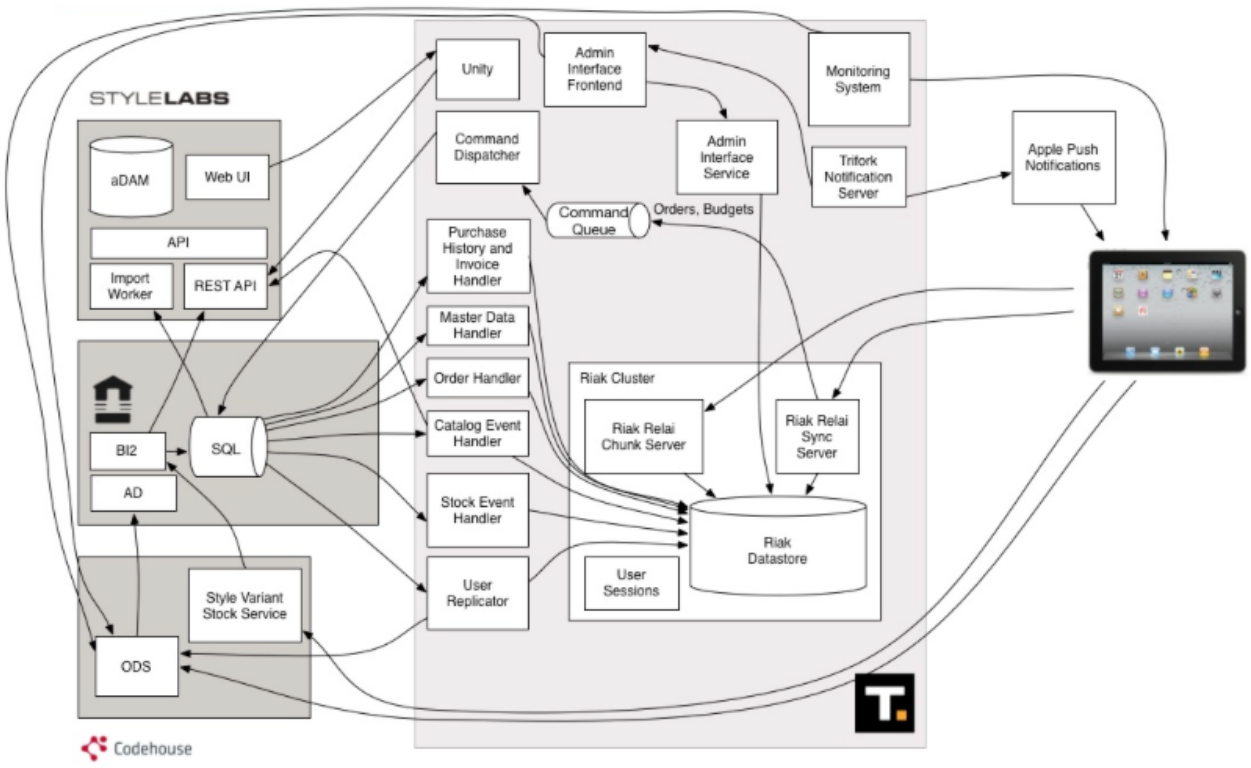

Figure 3: B2B eCommerce Architecture (P2tF).

References

[1] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 6:1–6:16, New York, NY, USA, 2015. ACM.

[2] Klaus Marius Hansen, Mads Ingstrup, Morten Kyng, and Jesper Wolff 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.

[3] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[4] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Rapport de recherche RR-7506, INRIA, January 2011. Printed.

[5] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and V. Villain, editors, *Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976, pages 386–400, Grenoble, France, October 2011.

[6] Surayya 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.

A TLA+ Representations of Use Cases

In this section, we present fully formal models of the following use cases.

A.1 Advertisement Counter

A.2 Leader Board

A.3 Virtual Wallet

A.4 Shared Medicine Record (FMK)

A.1 Advertisement Counter (AdCounter)

MODULE adCounterState

EXTENDS Naturals, GCounters

CONSTANTS
DV, Set of all devices.
DC, Set of all data centers.
maxTotalViews, Maximum number of times the ad should be shown.
maxTotalViewsPerDC, Maximum number of times the ad should be shown in a data center.
In this specification this partition is fixed.
maxTotalViewsPerDevice, Maximum number of times the ad should be shown in a device.
deviceAssignment Assignment of each device to a data center.
In this specification this assignment is fixed.

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

ASSUME $maxTotalViews \in Nat$
ASSUME $maxTotalViewsPerDevice \in [DV \to Nat]$
ASSUME $deviceAssignment \in [DV \to DC]$
ASSUME $maxTotalViewsPerDC \in [DC \to Nat]$
$\wedge\, SumAll([d \in DC \mapsto maxTotalViewsPerDC[d]]) = maxTotalViews$

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.

$State \triangleq$
[devices : SUBSET DV, Set of devices assigned to the data center.
maxViews : Nat, Maximum number of times the ad should be shown in

```
                    the data center. Used to transfer view rights.
    views : GCounter(DC),   G-Counter with overall views for the ad.

    viewsPerDevice : [DV → Nat]]  Number of views 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 ∈ [DC → State]

Init ≜
  ∧ TypeInvariant
  ∧ configuration =
     [d ∈ DC ↦ [devices ↦ {g ∈ DV : deviceAssignment[g] = d},
                maxViews ↦ maxTotalViewsPerDC[d],
                views ↦ GCounterInit(DC),
                viewsPerDevice ↦ [g ∈ DV ↦ 0]]]
```

```
Local operation at data center d that represents a visualisation of
the advert in device g at time t.
Pre: - Device g is assigned to data center d.
   - The data center view limit for the ad 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, g) ≜
  LET state ≜ configuration[d]
      gc ≜ state.views
      new_state ≜
         [devices ↦ state.devices,
          maxViews ↦ state.maxViews,
          views ↦ GCounterInc(gc, d),
          viewsPerDevice ↦ [state.viewsPerDevice EXCEPT ![g] = @ + 1]]
  IN   ∧ g ∈ state.devices
       ∧ GCounterValueAt(gc, d) < state.maxViews
       ∧ state.viewsPerDevice[g] < maxTotalViewsPerDevice[g]
       ∧ configuration′ = [configuration EXCEPT ![d] = new_state]
```

```
Merge(d1, d2) ≜
  LET state1 ≜ configuration[d1]
      state2 ≜ configuration[d2]
      new_state1 ≜
          [devices ↦ state1.devices,
           maxViews ↦ state1.maxViews,
           views ↦ GCounterMerge(state1.views, state2.views),
           viewsPerDevice ↦ state1.viewsPerDevice]
  IN  configuration′ = [configuration EXCEPT ![d1] = new_state1]

  Operation for consulting the number of views of ad a.
Views(d) ≜ SumAll(configuration[d].views)
```

```
Consistency ≜
    The ad views do not exceed the total views limite.
  ∧ ∀ d ∈ DC : Views(d) ≤ maxTotalViews

    The views local to a data center do not exceed limite for that data center.
  ∧ ∀ d ∈ DC : configuration[d].views[d] ≤ maxTotalViewsPerDC[d]

    The ad views of a device do not exceed the views limite for that device.
  ∧ ∀ d ∈ DC, g ∈ DV :
      configuration[d].viewsPerDevice[g] ≤ maxTotalViewsPerDevice[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.
  ∧ ∀ d ∈ DC, g ∈ DV :
     ∧ g ∉ configuration[d].devices ⟹ configuration[d].viewsPerDevice[g] = 0

    The ad views by devices matches the total ad views.
  ∧ ∀ d ∈ DC : SumAll(configuration[d].viewsPerDevice) = Views(d)

    GCounter property:
    The local value of a gcounter has to be greater or equal to the value in other.
  ∧ ∀ d1 ∈ DC, d2 ∈ DC :
      configuration[d1].views[d1] ≥ configuration[d2].views[d1]

Next ≜ ∃ d1 ∈ DC, d2 ∈ DC, g ∈ DV : Inc(d1, g) ∨ Merge(d1, d2)
```

vars $\triangleq$ ⟨configuration⟩

Spec $\triangleq$ Init $\wedge \Box[\text{Next}]_{\text{vars}}$

THEOREM Spec $\Longrightarrow$ TypeInvariant $\wedge$ Consistency

A.2 Leader Board

MODULE leaderBoard_v2

EXTENDS Naturals, TLC, FiniteSets
VARIABLES scores, vecclc
CONSTANTS DC, Games, Players, PlayersByGame

ASSUME PlayersByGame $\in$ [Games $\to$ SUBSET Players]

Function for converting Game-Player information function to tuple form

```
MapProduct(map) ≜
  LET mp[s ∈ SUBSET DOMAIN map] ≜
        IF s = {} THEN {}
                  ELSE LET y ≜ CHOOSE x ∈ s : TRUE
                       IN  ({y} × map[y]) ∪ mp[s \ {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 ∈ [DC → [MapProduct(PlayersByGame) → Nat]]
           ∧ vecclc ∈ [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 ∈ PlayersByGame[g] : ∀ y ∈ 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 ∈ 1 .. i] ≜ IF j = 1 THEN Leaders(d, g)
                              ELSE LET remaining ≜ PlayersByGame[g] \ RankSets[j − 1]
                                   IN  RankSets[j − 1] ∪
```

$$\{x \in \text{remaining} : \forall\, y \in \text{remaining} : \text{scores}[d][\langle g, x\rangle] \geq \text{scores}[d][\langle g, y\rangle]\}$$
$$\text{IN} \quad \text{IF } i = 1 \text{ THEN Leaders}(d, g) \text{ ELSE } \text{RankSets}[i] \setminus \text{RankSets}[i-1]$$

Initialize each field of every variable to zero

$$\text{Init} \triangleq \land \text{scores} = [d \in DC \mapsto [a \in \text{GamePlayers} \mapsto 0]]$$
$$\land \text{vecclc} = [g \in \text{GamePlayers} \mapsto [d \in DC \mapsto [dc \in DC \mapsto 0]]]$$

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

$$\text{UpdateScore}(dc, g, p, scr) \triangleq \land \langle g, p\rangle \in \text{GamePlayers}$$
$$\land scr > \text{scores}[dc][\langle g, p\rangle]$$
$$\land \text{scores}' = [\text{scores EXCEPT } ![dc][\langle g, p\rangle] = scr]$$
$$\land \text{vecclc}[\langle g, p\rangle][dc][dc] + 1 \in \text{Nat}$$
$$\land \text{vecclc}' = [\text{vecclc EXCEPT } ![\langle g, p\rangle][dc][dc] = \text{vecclc}[\langle g, p\rangle][dc][dc] + 1]$$

A helper function for finding maximum of two numerals

$$\text{Max}(n1, n2) \triangleq \text{IF } n1 \geq n2 \text{ THEN } n1 \text{ ELSE } n2$$

An operation that merges the score of player p in game g in data center d1 with the same field in data center d2
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 d2
POST : − update the field of the scores′ variable corresponding to player p for game g in data center d1 by merging 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 user for the same game in data center d2

$$\text{Merge}(d1, d2, g, p) \triangleq \land \langle g, p\rangle \in \text{GamePlayers}$$
$$\land \text{scores}[d2][\langle g, p\rangle] > \text{scores}[d1][\langle g, p\rangle]$$
$$\land \text{scores}' = [\text{scores EXCEPT } ![d1][\langle g, p\rangle] = \text{scores}[d2][\langle g, p\rangle]]$$
$$\land \text{LET new\_vc} \triangleq [d \in DC \mapsto \text{Max}(\text{vecclc}[\langle g, p\rangle][d1][d], \text{vecclc}[\langle g, p\rangle][d2][d])]$$
$$\text{IN} \quad \text{vecclc}' = [\text{vecclc EXCEPT } ![\langle g, p\rangle][d1] = \text{new\_vc}]$$
$$\land \text{LET a} \triangleq \langle \text{“MERGE”}, d1, g, 2, \text{Rank}(d1, g, 2), \text{scores}\rangle \text{IN} \quad \text{Print}(a, \text{TRUE})$$

$$\text{Next} \triangleq \exists\, d \in DC, d2 \in DC, g \in \text{Games}, p \in \text{Players}, scr \in \text{Nat} : (\text{UpdateScore}(d, g, p, scr) \lor \text{Merge}(d, d2, g, p))$$

$$\text{Spec} \triangleq \text{Init} \land \Box[\text{Next}]_{\langle \text{scores, vecclc}\rangle}$$

Eventual Consistency invariant stating that scores variable eventually converges

$$\text{Convergence} \triangleq \Diamond\forall\, d1 \in DC,\ d2 \in DC,\ g \in Games,\ p \in Players :$$
$$\langle g,\ p\rangle \in GamePlayers \implies scores[d1][\langle g,\ p\rangle] = scores[d2][\langle g,\ p\rangle]$$

if the vector clock for all players playing game g in d1 is $\leq$ those of the same players for game g in data center d2 then Leader's score for g in d1 must be $\geq$ Leader's score for g in d2

$$\text{MonotonicLeader} \triangleq \forall\, d1 \in DC,\ d2 \in DC,\ g \in Games :$$
$$(\forall\, p \in Players : \langle g,\ p\rangle \in GamePlayers \implies$$
$$(\forall\, d \in DC : vecclc[\langle g,\ p\rangle][d1][d] \leq vecclc[\langle g,\ p\rangle][d2][d])) \implies$$
$$scores[d1][\langle g,\ \text{CHOOSE } x \in Leaders(d1,\ g) : \text{TRUE}\rangle] \leq$$
$$scores[d2][\langle g,\ \text{CHOOSE } x \in Leaders(d2,\ g) : \text{TRUE}\rangle]$$

A.3 Virtual Wallet

A.3.1 Virtual Walet using CRDTs and Transactions

MODULE walletv4

EXTENDS Naturals, Sequences, TLC

VARIABLES wallets

CONSTANTS Replicas, V1Cost, InitBal, Natlim, Qtylim

ASSUME $\land$ V1Cost $\in$ Nat
$\land$ InitBal $\in$ Nat
$\land$ Natlim $\in$ Nat
$\land$ Qtylim $\in$ Nat
$\land$ V1Cost > 0
$\land$ Natlim > 0
$\land$ Qtylim > 0

$$PNCounter(dom) \triangleq [p : [dom \to Nat],\ n : [dom \to Nat]]$$

$$InitPNCounter(dom) \triangleq [p \mapsto [d \in dom \mapsto 0],\ n \mapsto [d \in dom \mapsto 0]]$$

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

```
                                                IN  map[y] + Sum[r \ {y}]
    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 ∈ DOMAIN pnc1.p ↦ Max(pnc1.p[d], pnc2.p[d])],
              n ↦ [d ∈ DOMAIN pnc1.n ↦ Max(pnc1.n[d], pnc2.n[d])]]
```

```
Wallet ≜ [balance : PNCounter(Replicas),
          v1cnt : PNCounter(Replicas),
          vecclc : [Replicas → Nat]]

TypeInv ≜  ∧ wallets ∈ [Replicas → Seq(Wallet)]

Init ≜   ∧ Print ("a", TRUE)
         ∧ wallets = [r ∈ Replicas ↦ ⟨[balance ↦ InitPNCounter(Replicas),
                                       v1cnt  ↦ InitPNCounter(Replicas),
                                       vecclc  ↦ [r2 ∈ Replicas ↦ 0]]⟩]

App(elt, s) ≜ ⟨elt⟩ ∘ 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 ∈ 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  ∧ ∃ r ∈ 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 ∈ 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  ∧ ∃ r ∈ Replicas : wr1.vecclc[r] < wr2.vecclc[r]
      ∧ wallets′ = [wallets EXCEPT ![rep1] = App(wr1_new, wallets[rep1])]
      ∧ Print("MergeLastStates", TRUE)
       ∧ Print(wallets′, TRUE)
```

In each replica, money spent = number of vouchers bought x unit cost of voucher

```
ConservationOfMoney ≜ ∀ r ∈ Replicas : EvalPNCounter(Head(wallets[r]).balance) +
    EvalPNCounter(Head(wallets[r]).v1cnt) ∗ V1Cost = 0
```

Balance in the wallet is always positive

```
PosBalance ≜ ∀ rep ∈ 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 ≜  ∧ ∀ r ∈ Replicas, r2 ∈ Replicas, i ∈ Nat : (i > 0 ∧ i < Len(wallets[r])) ⟹
    (GetElt(wallets[r], i).vecclc[r2] ≥ GetElt(wallets[r], i + 1).vecclc[r2]
```

$\land$ GetElt(wallets[r], i).balance.p[r2] $\geq$ GetElt(wallets[r], i + 1).balance.p[r2]
$\land$ GetElt(wallets[r], i).balance.n[r2] $\geq$ GetElt(wallets[r], i + 1).balance.n[r2]
$\land$ GetElt(wallets[r], i).v1cnt.p[r2] $\geq$ GetElt(wallets[r], i + 1).v1cnt.p[r2]
$\land$ GetElt(wallets[r], i).v1cnt.n[r2] $\geq$ GetElt(wallets[r], i + 1).v1cnt.n[r2])

FinalState(vc) $\triangleq$ $\forall$ rep $\in$ Replicas : vc[rep] = Natlim
EqualStates(st1, st2) $\triangleq$ $\forall$ rep $\in$ Replicas : st1.balance.p[rep] = st2.balance.p[rep] $\land$
st1.balance.n[rep] = st2.balance.n[rep] $\land$
st1.v1cnt.p[rep] = st2.v1cnt.p[rep] $\land$
st1.v1cnt.n[rep] = st2.v1cnt.n[rep]

Eventually all states converge to the same state

Convergence $\triangleq$ $\forall$ r1 $\in$ Replicas, r2 $\in$ Replicas :
FinalState(Head(wallets[r1]).vecclc) $\land$ FinalState(Head(wallets[r2]).vecclc) $\Longrightarrow$
EqualStates(Head(wallets[r1]), Head(wallets[r2]))

Next $\triangleq$ $\exists$ r1 $\in$ Replicas, r2 $\in$ Replicas, qty $\in$ 1 .. Qtylim , i $\in$ Nat :
(BuyV1(r1, qty) $\lor$ Merge(r1, r2, i)MergeLastStates(r1, r2))

Spec $\triangleq$ Init $\land$ $\Box$[Next]$_{\langle \text{wallets} \rangle}$

THEOREM Spec $\Longrightarrow$ TypeInv $\land$ ConservationOfMoney

A.3.2 Virtual Walet 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

Wallet $\triangleq$ [balance : Int,
v1cnt : Int,

```
            vecclc : [Replicas → Nat]]

TypeInv ≜  ∧ wallets ∈ [Replicas → Wallet]

Init ≜   ∧ wallets = [r ∈ Replicas ↦ [balance ↦ InitBal,
                                      v1cnt ↦ 0,
                                      vecclc ↦ [r2 ∈ 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]

Max(n1, n2) ≜ IF n1 ≥ n2 THEN n1 ELSE  n2

Merge(r1, r2) ≜
   LET wr1 ≜ wallets[r1]
       wr2 ≜ wallets[r2]
       new_vc ≜ [r ∈ Replicas ↦ Max(wr1.vecclc[r], wr2.vecclc[r])]
       new_w1 ≜ [balance ↦ Max(wr1.balance, wr2.balance),
                 v1cnt ↦ Max(wr1.v1cnt, wr2.v1cnt),   alternative formulation: wr1.v1cnt + wr2.v1cnt
                 vecclc ↦ new_vc]

   IN   ∧ ∃ r ∈ Replicas : wr1.vecclc[r] < wr2.vecclc[r]
        ∧ wallets′ = [wallets EXCEPT ![r1] = new_w1]


ConservationOfMoney ≜ ∀ r ∈ Replicas : (InitBal − wallets[r].balance) ≥ wallets[r].v1cnt ∗ V1Cost

FinalState(vc) ≜ ∀ r ∈ Replicas : vc[r] = Natlim
EqualStates(st1, st2) ≜ st1.balance = st2.balance ∧ st1.v1cnt = st2.v1cnt
Convergence ≜ ∀ r1 ∈ Replicas, r2 ∈ Replicas :
      FinalState(wallets[r1].vecclc) ∧ FinalState(wallets[r2].vecclc) ⟹
            EqualStates(wallets[r1], wallets[r2])
```

$\text{Next} \triangleq \exists\, \text{r1} \in \text{Replicas}, \text{r2} \in \text{Replicas}, \text{qty} \in 1 \,..\, \text{Qtylim} : (\text{BuyV1}(\text{r1}, \text{qty}) \lor \text{Merge}(\text{r1}, \text{r2}))$

$\text{Spec} \triangleq \text{Init} \land \Box[\text{Next}]_{\langle \text{wallets} \rangle}$

THEOREM $\text{Spec} \implies \text{TypeInv} \land \text{ConservationOfMoney} \land \text{Convergence}$

A.3.3 Wallet Use Case without Transactions

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 $\land \text{V1Cost} \in \text{Nat}$
$\land \text{InitBal} \in \text{Nat}$
$\land \text{Natlim} \in \text{Nat}$
$\land \text{Qtylim} \in \text{Nat}$
$\land \text{V1Cost} > 0$
$\land \text{Natlim} > 0$
$\land \text{Qtylim} > 0$

$\text{PNCounter}(\text{dom}) \triangleq [\text{p} : [\text{dom} \to \text{Nat}], \text{n} : [\text{dom} \to \text{Nat}]]$

$\text{InitPNCounter}(\text{dom}) \triangleq [\text{p} \mapsto [\text{d} \in \text{dom} \mapsto 0], \text{n} \mapsto [\text{d} \in \text{dom} \mapsto 0]]$

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

$\text{EvalPNCounter}(\text{pnc}) \triangleq \text{SumAll}(\text{pnc.p}) - \text{SumAll}(\text{pnc.n})$

```
Max(n1, n2) ≜ IF n1 ≥ n2 THEN n1 ELSE n2

MergePNCounters(pnc1, pnc2) ≜ [p ↦ [d ∈ DOMAIN pnc1.p ↦ Max(pnc1.p[d], pnc2.p[d])],
                               n ↦ [d ∈ DOMAIN pnc1.n ↦ Max(pnc1.n[d], pnc2.n[d])]]
```

```
Wallet ≜ [balance : PNCounter(Replicas),
          v1cnt : PNCounter(Replicas),
          vecclc : [Replicas → Nat]]

TypeInv ≜ ∧ wallets ∈ [Replicas → Wallet]

Init ≜
    ∧ wallets = [r ∈ Replicas ↦ [balance ↦ InitPNCounter(Replicas),
                                 v1cnt   ↦ InitPNCounter(Replicas),
                                 vecclc  ↦ [r2 ∈ 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)

Merge(rep1, rep2) ≜
  LET wr1 ≜ wallets[rep1]
      wr2 ≜ wallets[rep2]
      new_vc ≜ [r ∈ 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   ∧ ∃ r ∈ Replicas : wr1.vecclc[r] < wr2.vecclc[r]
       ∧ wallets′ = [wallets EXCEPT ![rep1] = wr1_new]
       ∧ Print("MergeLastStates", TRUE)

MergeBalance(rep1, rep2) ≜
  LET wr1 ≜ wallets[rep1]
      wr2 ≜ wallets[rep2]
      new_vc ≜ [r ∈ Replicas ↦ Max(wr1.vecclc[r], wr2.vecclc[r])]
      wr1_new ≜ [balance ↦ MergePNCounters(wr1.balance, wr2.balance),
                 v1cnt ↦ wr1.v1cnt,
                 vecclc ↦ new_vc]
  IN   ∧ ∃ r ∈ Replicas : wr1.balance.n[r]   < wr2.balance.n[r]
       ∧ wallets′ = [wallets EXCEPT ![rep1] = wr1_new]
       ∧ Print("MergeBalance", TRUE)

MergeCount(rep1, rep2) ≜
  LET wr1 ≜ wallets[rep1]
      wr2 ≜ wallets[rep2]
      new_vc ≜ [r ∈ Replicas ↦ Max(wr1.vecclc[r], wr2.vecclc[r])]
      wr1_new ≜ [balance ↦ wr1.balance,
                 v1cnt ↦ MergePNCounters(wr1.v1cnt, wr2.v1cnt),
                 vecclc ↦ new_vc]
  IN   ∧ ∃ r ∈ Replicas : wr1.v1cnt.p[r] < wr2.v1cnt.p[r]
       ∧ wallets′ = [wallets EXCEPT ![rep1] = wr1_new]
       ∧ Print("MergeBalance", TRUE)
```

In each replica, money spent = number of vouchers bought x unit cost of voucher

```
ConservationOfMoney ≜ ∀ r ∈ Replicas :
      EvalPNCounter(wallets[r].balance) + EvalPNCounter(wallets[r].v1cnt) ∗ V1Cost = 0
```

Balance in the wallet is always positive — DOES NOT HOLD

```
PosBalance ≜ ∀ rep ∈ Replicas : InitBal + EvalPNCounter(wallets[rep].balance) ≥ 0

FinalState(vc) ≜ ∀ rep ∈ Replicas : vc[rep] = Natlim
EqualStates(st1, st2) ≜ ∀ rep ∈ Replicas :
      st1.balance.p[rep] = st2.balance.p[rep] ∧
```

st1.balance.n[rep] = st2.balance.n[rep] ∧
st1.v1cnt.p[rep] = st2.v1cnt.p[rep] ∧
st1.v1cnt.n[rep] = st2.v1cnt.n[rep]

Eventually all states converge to the same state — DOES NOT HOLD

Convergence $\triangleq$ ∀ r1 ∈ Replicas, r2 ∈ Replicas :
FinalState(wallets[r1].vecclc) ∧ FinalState(wallets[r2].vecclc) ⟹
EqualStates(wallets[r1], wallets[r2])

Next $\triangleq$ ∃ r1 ∈ Replicas, r2 ∈ Replicas, qty ∈ 1 .. Qtylim :
(BuyV1(r1, qty) ∨ MergeCount(r1, r2) ∨ MergeBalance(r1, r2))

Spec $\triangleq$ Init ∧ □[Next]$_{\langle \text{wallets} \rangle}$

THEOREM Spec ⟹ TypeInv ∧ ConservationOfMoney ∧ PosBalance ∧ Convergence

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 ∈ Nat

TimeStamps $\triangleq$ DC × (1 .. MAX)

Initialize Variables

Init $\triangleq$
∧ patientdb = [d ∈ DC ↦ [p ∈ Pat ↦ [treat ↦ {},
presc ↦ {},
taken ↦ {}]]]

$\wedge$ clock = [d $\in$ DC $\mapsto$ 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) $\triangleq$
LET
timestamp $\triangleq$ $\langle$dc, clock[dc] + 1$\rangle$
s $\triangleq$ patientdb[dc]
t $\triangleq$ $\langle$doctor, treatment, timestamp$\rangle$
IN
$\wedge$ patientdb$'$ = [patientdb EXCEPT ![dc][patient].treat = s[patient].treat $\cup$ {t}]
$\wedge$ clock$'$ = [clock EXCEPT ![dc] = clock[dc] + 1]

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

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

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

giveDrug(dc, patient, doctor, treatment, timestamp, prescription, pharmacy) $\triangleq$
LET
timestamp2 $\triangleq$ $\langle$dc, clock[dc] + 1$\rangle$
p $\triangleq$ $\langle$doctor, treatment, timestamp, prescription, pharmacy, timestamp2$\rangle$

```
        s ≜ patientdb[dc]
    IN
        ∧ ⟨doctor, treatment, timestamp, prescription⟩ ∈ s[patient].presc
    ∧ ∃ ts ∈ TimeStamps : ⟨doctor, treatment, timestamp, prescription, pharmacy, ts⟩ ∈ s[patient].ta
    ∧ patientdb′ = [patientdb EXCEPT ![dc][patient].taken = s[patient].taken ∪ {p}]
        ∧ clock′ = [clock EXCEPT ![dc] = clock[dc] + 1]

 Merge two datacenter databases
merge(dc1, dc2) ≜
  LET
    s1 ≜ patientdb[dc1]
    s2 ≜ patientdb[dc2]
    ns ≜ [p ∈ Pat ↦ [treat ↦ s1[p].treat ∪ s2[p].treat,
                     presc ↦ s1[p].presc ∪ s2[p].presc,
                     taken ↦ s1[p].taken ∪ s2[p].taken]]
  IN
    ∧ patientdb′ = [patientdb EXCEPT ![dc1] = ns, ![dc2] = ns]
    ∧ UNCHANGED ⟨clock⟩

NoDrugGivenTwice ≜ ∀ d ∈ DC : ∃ p ∈ Pat : ∃ t1 ∈ patientdb[d][p].taken, t2 ∈ patientdb[d][p].taken :
  t1 ≠ t2 ∧ SubSeq(t1, 1, 5) = SubSeq(t2, 1, 5)      DO NOT HOLD

Next ≜
  ∨ ∃ dc ∈ DC, patient ∈ Pat, doctor ∈ Doc, treatment ∈ Tre :
                              addTreatment(dc, patient, doctor, treatment)
  ∨ ∃ dc ∈ DC, patient ∈ Pat, doctor ∈ Doc, treatment ∈ Tre, ts ∈ TimeStamps, prescription ∈ Pre :
      addPrescription(dc, patient, doctor, treatment, ts, prescription)
  ∨ ∃ dc ∈ DC, patient ∈ Pat, doctor ∈ Doc, treatment ∈ Tre,
    ts ∈ TimeStamps, prescription ∈ Pre, pharmacy ∈ Pha :
      giveDrug(dc, patient, doctor, treatment, ts, prescription, pharmacy)

Spec ≜ Init ∧    □[Next]⟨patientdb, clock⟩

THEOREM Spec ⟹ □NoDrugGivenTwice
```