Real Life Security - Security Protocols Case Study with the Downsizing Game

Théo Dubourg

August 12, 2014

Abstract

Real life security use cases often include multiple parties talking to each other and making basic contracts between each other. A contract can take several forms: A contract between two programs about how they are going to communicate, a contract between two business men about a deal between their respective companies, etc. . In this paper, we are going to focus on contracts that can be modeled as *transactions* sequences. More specifically, we are going to do a case study using the Downsizing Game invented by [1] . The Downsizing Game puts in place multiple players that are all pursuing the same goal of maximizing their own profit. Two players can make transactions between each other in order to trade every sort of resources that is available in the game. The goal of the case study is to see what are the steps to put in place a judging party that will enforce security protocols related to cheating prevention and what are the security requirements we can add on top of the functional requirements. Throughout our work setting up such a system and environment, we will report on the issues we face, the solutions we find and the decision we make with the justifications for such choices.

Keywords: enforcement, security, security protocols

1 Introduction

1.1 The Downsizing Game

The Downsizing Game, as originally described by S. Kaitani [1] , puts multiple players into an environment where they are given a million of the local currency and have to return it at the end of the game, but they can keep the remainder if they managed to make some profit. There are voting rounds where they vote for the other players. The player getting the maximum amount of votes wins a one million prize while the one with the least amount of votes is removed from the group (the group is *downsized*, giving the name to the game).

Players can maximize their profit either by trading resources with other players, or win the one million prize.

Two players can make transactions between each other in order to trade every sort of resources that is available in the game: Money, trust, loyalty, and votes.

The game is overseen by a *judging party*, also called *game master*. Its role in the original game is to make sure that nobody is cheating and everything in the game happens according to the game's rules. He validates transactions and he applies fines equal to the amount of the starting money when players do not respect their transactions.

We will describe the exact instance of the game that we will use (with variations in the rules) in details in section 2.2.

Although it may seem like a simple game, it poses interesting challenges in terms of security because players will almost certainly try to cheat in order to win the game and maximize their profit.

Example

We will take a short example on a very small instance of the Downsizing Game to show the game's dynamics.

Note that our instance used for the case study and described in section 2.2 will differ in some details to this example (voting rounds, especially), but we keep the example simplistic for now.

Let us imagine we have 3 players, 6 rounds, 3 voting rounds (every 2 rounds) and every player can vote for only one other player on each voting round.

The following actions are one of the possible runs of the game (currency is $ here):

1. Player1 is the current player. Player1 sells 1 vote to Player2 for $100,000.
2. Voting round. Player2 votes for player 1 to respect the transaction. Player3 votes for Player2, Player1 votes for Player3. Scores are now 1 for everyone.
3. Player2 is the current player and buys Player1 1 vote and Player3 1 vote for $25,000 each.
4. Voting round. Player2 receives 2 votes from players 1 and 3. Player3 receives a vote from Player2. Scores are now respectively 1, 3 and 2 for players 1, 2 and 3.
5. Player3 tries to buy votes from the two other players for $500,000 each, Player2 accepts but Player1 refuses.
6. Voting round. Player2 votes for Player3 to respect the transaction. Player3 votes for Player1. Player1 finally still votes for Player3. Scores are now respectively respectively 2, 3 and 4 for players 1, 2 and 3.

The Player3 is the only winner and gets the one million prize. All players then have to give back the orignal one million. Player1 cannot and has a debt of $75,000. Player2 gives it back and keeps the $450,000 remaining. Player3 has a debt of $475,000 but wins the one million prize and thus make a profit of $525,000 in the end.

1.2 Objective

The objective, in this paper, is to use the Downsizing Game as an example to study the process of a small-sized software development project that would start with functional requirements and then add security requirements.

We want to build a platform to allow players to play the Downsizing Game but we do not want any cheaters.

Throughout our design and implementation, we will report on the issues we faced and the solutions and decisions we took to overcome them, along with the justifications of such choices. We think this material can be of some interest to computer science students or beginners in providing them with a concrete and detailed example.

The work is split into 3 main steps:

1. Functional requirements: design & implementation
2. Security requirements: design & implementation
3. Comparison against existing coding guidelines

In step 2 above (*Security requirements: design & implementation*), we will define and describe security requirements that we found, by intuition, necessary to be added.

By "by intuition", we mean here: simply thinking about the different cases that will be faced in the game and what could be attempted by player to cheat on the game and what should be done to ensure those cheating attempts will fail.

In step 3 above (*Comparison against existing coding guidelines*), we will compare our already security-aware (thanks to *step 2*) application to coding guidelines. We will conclude on what we were able to fulfill or not of those guidelines by ourselves (that is to say, before reading them). We will also analyze what, in the guidelines, applies to our program and what does not.

We will look at the last step as a partial answer to the question "How secure can I code?".

2 Case Study

2.1 Assumptions

In our design and implementation of the Downsizing Game, we will start from the following assumptions:

1. The communication channels, e.g. between the players and the judging party (section 2.2), are secure.
2. Everything is happening in the same process on a single CPU machine. That is to say: only one instruction can be executed at a time, there is no parallelism/concurrency.
3. The technology used to develop the game prevents arbitrary memory from being read. A player program will thus only be allowed to access data from objects it has been given an explicit pointer and/or reference to.

The reason for working under those assumptions is that we focus here on the security of the interaction protocol between players and the implementation of the judging party. A communication layer would bring with it all sorts of communication-related security concerns that are not of interest here. The same reason goes for parallelism and/or concurrency.

2.2 Functional Requirements

The functional requirements are the set of features that our instance of the Downsizing Game should implement/respect.

Game's rules

The rules of *our instance* of the Downsizing Game will be stated as follows:

- The game has a fixed length of 1,000 rounds.
- 3 players participate in the game.
- Players can only participate once in the game.
- Valid/tradable *resources* are votes, score and the game currency.
- Every player is given a million units of the game currency (let us call it dollars) and 10 votes to cast.
- Every player starts with a score of zero.
- At the end of the game, every player must return the original one million dollars that she was given in the beginning, she can keep the remainder of the money for her.
- At the end of the game, if players cannot give back the entire amount of money that they were given in the beginning, they have contracted a debt that they will have to reimburse.
- Two players can make transactions between each other using every available resource in the game.
- A judging party enforces the game's rules and manages transactions, ensuring their validity and preventing players from cheating.
- There are 10 voting rounds: rounds 100, 200, 300, 400, 500, 600, 700, 800, 900 and 1,000.
- All players must vote on voting rounds.
- Players must cast exactly 1 vote at every voting rounds.
- Players can not vote for themselves.
- Every time a player receives a vote, his *score* increases by 1.
- At the end of the game, the player(s) with the highest score win(s). The one(s) with the lowest score lose(s).
- The winning player(s) earn(s) 1 million dollars (evenly split, if multiple winners).
- If a player is banned (because of cheating), she has to return the money she was given at the beginning as well. If she cannot, she is considered as having a debt, the samw way as if she had finished the game with a debt.

As a consequence of these rules, here are some examples of possible basic strategies:

- A player can try to get the maximum amount of votes, in order to be the winner, thus maximizing profit by getting the winning prize.
- A player can try to maximize its profit, without caring about votes. She will not win, but will still make profit out of the game, as she will keep the remainder of the money after giving back the original one million dollars.

Definitions & needed features

Judging party

The judging party is a neutral (not affiliated with any player) entity whose responsability is to maintain order in the game and make sure nobody is able to cheat and that, to the extent of its knowledge, every transaction is valid.

Current player

Players play on a turn-by-turn basis. We will call *current player* the player of which it is currently the turn.

Rounds

A *round* is an atomic unit of time in the game. A round can thus not be divided into subrounds. A round passes every time any of the following actions is executed:

- Applying a transaction successfully
- Changing the current player
- Applying the result of a voting round

Voting rounds

A voting round is a round where, **before** any action from a player, all players will be asked to vote as described in the game's rules.

According to the definition of the rounds, no other actions can be thus taken during a voting round (after the votes casting), as a round passes when applying the result of a voting round.

Vote

A *vote* is a special resource that can only be transferred during a voting round, by vote casting (section 3.1).

Resources

A *resource* is an integer quantity that the judging party allows to trade. In our case study, the set of resources will be fixed at the beginning of the game, to:

- Cash/money/currency
- Votes
- Score

Score

The *score* of a player is increased by 1 for every vote that is casted from another player for this player.

Players can also trade their score as a *resource*. The initial score is zero (see rules (section 2.2)). Their *current balance* of the *score resource* is thus their *current score*.

Cheater

A cheater is a player that breaks one or several game's rules. Cheaters must be immediately banned and thus removed from the game. Banned players cannot participate in the game anymore. As such, they will not be able to use the game again to purge their debt if they have some.

Transactions

A *transaction* is one or multiple transfer(s) of *fixed amounts* of resources between two *identified/authenticated* (see section 2.3) players. The player *sending* the resource will later be referred to either as the *sending player* or *paying player* or *payer* or *sender*. The player *receiving* the resource will later be referred to either as the *receiving player* or *paid player* or *payee* or *recipient*.

A transaction can either be unidirectional, that is to say, a player transfers some resource to another player and that is all, or bidirectional. In the latter case, two players transfer resources to each other.

A bidirectional transaction is composed (in the OOP meaning) of two unidirectional transactions. A bidirectional is **not** a special unidirectional transaction (it does not inherit from it).

Every resource except *voting promises* can be traded using either an unidirectional or bidirectional transactions. Voting promises have to be traded using *voting promises transactions* (section 2.2).

Amount

An *amount* is a defined, positive integer, quantity of a single resource.

Immediate transactions

Immediate transactions are the basic transactions: As soon as the transaction is applied (after being validated), the transfer(s) of resources is/are applied.

As a consequence, when a player agrees on an *immediate* transaction, she is assured that the transaction will be fulfilled if it's validated by the judging party (and thus applied, as validated transaction are then applied).

Delayed/scheduled transactions

Scheduled or *delayed* transactions are transactions where some transfer(s) of resources is/are not immediate.

A delayed transaction is a transaction with an additional information about an **absolute game time unit**. The amount in this *delayed* transaction has to be completely transferred (strictly) *before* this absolute game time unit.

Just like immediate transactions, delayed transactions can be either *unidirectional* or *bidirectional*. A bidirectional transaction is said to be *delayed* or *scheduled* if and only if at least one of the two unidirectional transactions it is composed of, is a delayed transaction.

Voting promises transactions

Voting promises are a type of delayed transactions. Voting promises are promises that a given player will cast a given number of votes for the *recipient* of the transaction before a given absolute game's time unit (because it's a delayed transaction).

Voting Transactions

Voting transactions are *immediate unidirectional* transactions. Voting transactions can only be instantiated by the judging party. The judging party will instantiate a *voting transaction* for every valid vote during a *voting round* (section 2.2). And will apply them all at the end of the voting round.

Voting transactions are a mechanism that will allow to check for the fulfillment of *voting promises transactions*. The details will be presented with the transactions validation details (section 3.1).

Valid transaction

A *transaction* is said to be *valid* if and only if:

- Transaction does not break a game rule.
- Transaction is able to be completed to the extent of the judging party's knowledge. (e.g.: Enough money on the account, in case of an immediate money transfer).

A bidirectional transaction will be considered valid if both unidirectional transactions it is composed of are valid. If any of them is not, then the bidirectional transaction is also invalid.

2.3 Security Requirements

Now that we have defined functional requirements, we are going to define and describe the additional *security requirements* that are necessary to ensure the players do not exploit singularities of the game in order to achieve behaviours that should not be achieved according to the rules.

Game state alteration

The judging party must be the only one allowed to alter the state of the game (including applying transactions). Players talk to it when they want to make a transaction and it validates (or invalidates) it and applies it (or refuses it).

The *game state* here refers to the balances of all players (including scores), the current round number, the history of transactions and the state of all transactions in the history.

Judging party neutrality

The judging party must not be manipulated by players in any way in order to achieve its neutrality. Players should not be able to alter the state of the judging party or tamper with any of its internal data.

Resources accounting protection

We cannot let the players update by themselves the amount of resources as they would obviously cheat and generate for instance new cash flow so that they keep buying and making profits. We should have something managing the resources accounting.

Judging party exclusiveness aka turn-by-turn enforcement

The *current player* must be the only one that can make calls to the judging party's interface.

This is needed in order to prevent other players from trying to steal the CPU and make transactions on the transaction quota of another player instead of their own quota. We should thus implement a way to protect the judging party from executing actions calls by other players than the *current player*.

Immediate transactions: guaranteeing "immediateness" (atomicity)

There should be no other interactions, no changes to the state of the game should happen between validation and application of an immediate transaction.

This is partially guaranteed by the assumptions of no parallelism/concurrency. Although, a player program could still manage to steal the CPU (effectively pausing the judging party program's execution) and try to change the state of the game by submitting other transactions for instance. A player could use such an attack to submit multiple transactions that will be validated on the same balances, but executed on different balances (race-condition/ "TOCTTOU" [3]).

Delayed/scheduled transactions completion

Delayed transactions are validated without the transfers of resources actually taking place and have a deadline for this transfer to happen. There should a mechanism in place to check that the transfer was indeed done strictly before the deadline.

Players authentication

When creating a transaction, a player could provide false information about the participants in this transaction and thus attempt to impersonate other players and make transactions on their behalf.

As a consequence, we need a mechanism to authenticate the players and be sure that the player who submitted a request in her name, is indeed the player she is saying she is.

3 Implementation

In this section, we will first describe parts of the implementation related to enforcing the *security requirements* and then present the remaining of the implementation's details.

3.1 Security requirements implementation

Resources accounting

The accounting of the resources of all players will entirely be hidden from the players and be done exclusively by the judging party.

Players will be given their initial amount of resources and it will be left up to them to then track their resources if they need to. The judging party's interface will not include anything to read the resources/balances of any players.

The judging party, before the instantiation of the players, will instantiate every player's resources. As every transaction has to go through the judging party, it will also be the one responsible for updating the resources balances.

This way, we do not need any access control mechanism for the players, they do not have any knowledge about the "real" accounts and as a consequence, no access to it (as they cannot access objects without a reference to them, as per section 2.1)

Judging party exclusiveness aka turn-by-turn enforcement

In order to allow only the current player to access the judging party and make calls to it, we will simply keep track of the current player *id* in the judging party and every judging party's interface function will check whether the calling player is the current player or not and immediately abort if not. Player authentication (section 2.3) guarantees that another player cannot submit transactions by using the *current player* as the origin of the transaction.

Transaction History Log

In order to keep track of every transaction that happened in the game (and to allow for *completion check* (section 3.1)), the judging party will keep updated a transaction *transaction history* log. This log will contain absolutely all transactions that have been *validated* by the judging party together with the round number they were validated at.

Immediate transactions: guaranteeing "immediateness" (atomicity)

As explained in section 2.3, even with the stated assumptions (section 2.1), there is still room for a race-condition.

To avoid this, we will make use of synchronization tools to only allow one concurrent execution of the judging party program's code of validation and application of an immediate transaction. This can be materialized for instance by the use of operating system's *mutex* tools, but any other synchronization tool providing equal guarantees can be used too.

Delayed/scheduled Transactions Completion Check

When the game's clock ticks to this absolute time unit (the deadline), it will tell the judge that there is some delayed transactions that should be checked for having been completed. The judging party will then check if the transactions have been completed by the players participating in the scheduled transaction. Checks for completion of delayed transactions will **always** be performed **before any other action** can be taken by any player in the current round. If one or both players that was/were supposed to transfer the resources did not transfer the exact amount of resources she/they was supposed to, this/these player(s) will be considered a cheater/cheaters.

Note that in order to guarantee that the current player is not able to talk to the judging party before the end of the delayed transactions checks, we will once more make use of synchronization tools, as in the case of immediate transactions atomicity (section 2.3).

But in order to be able to tell the difference between transfers of resources related to new deals between two players and transfers of resources related to completion of a scheduled/ delayed transaction, we need an additional, special type of transaction. We will call it "subtransaction".

A **subtransaction** is an **immediate unidirectional** transaction that contains an additional information about the delayed transaction is it related to. It has to be of the same *resource type* as the delayed transaction it is related to. It cannot be refused by the *recipient*. Indeed, if we allowed the recipient to refuse it, one could force someone into being banned from the game by agreeing upon a delayed transaction and then refusing the subsequent substransactions made to complete it, by the other player.

When the judging party has to check that the delayed transaction was indeed completed, it will go through all subtransactions that were made since the round of the delayed transaction between the same two players and sum everything up. The sum has to be **exactly** what was initially agreed in the delayed transaction. If and only if it is the case, then the transaction can be marked as completed.

Voting Promises Completion Check

Voting promises, as defined in section 2.2, are *delayed transactions* as well. They thus need to be checked for completion too. These transactions will be part of the "transactions to be checked" that will be given to the judging party at the very beginning of a round (the round that is the transaction's deadline) as presented in the previous section. When the judging party is given a *voting promise transaction* to check completion for, it will not look for *subtransactions* in order to check that the amount of *votes* was transferred. Instead, it will look for *voting transactions*. *Voting transactions* mechanism is presented in details in section 3.1.

Players authentication

To authenticate the player, we will go for the simple use of an id and a password. At the beginning of the game, every player will be given a unique password, together with her *id*. She will have to pass them along with every call she makes to the judging party's interface in order to prove this call is coming from the player it is said it comes from.

When the judging party calls players itself, the authentication is assured as the judging party has direct pointers to every players and those pointers cannot be tampered by the other players (as per assumption 3 (section 2.1))

Transactions validation

When asked to perform a new transaction, the judging party will sequentially (in the order below) go through multiples steps, or checks.

At every step, if the check fails, the transaction is marked as not valid, and thus, refused.

1. Players mutual agreement

For any transaction validation, the first step that the judge will follow is to ask both involved players whether they confirm that they agree with this transaction. Both players have to answer "yes" for this check to succeed.

Note that if the transaction if a *voting transaction* (vote casting), only the agreement of the *payer* is being asked. The *payee* is not asked because you cannot *refuse* a vote.

2. Input validation

The judge will then perform rational checks. These checks are the following:

- Is the amount a **positive integer**? (avoiding resources "generation" with negative values, and rounding errors exploits if we were to use real numbers)
- Is the amount smaller than the global amount of resources of this type in the entire game?
- If it is a delayed transaction, is the *deadline* a valid round? ($current\ round < deadline \leq last\ round$). Note that this check means that no *delayed transaction* can be made during the last round.
- In case of a voting promise, is the amount smaller than the number of votes the player will be able to cast before the end of the game? (remaining voting rounds multiplied by votes per player and per voting round)

3. Balance check

The judge will then check the balance of the player *from* which the transfer is going to happen for the resource that is going to be transferred.

In the case of immediate transactions, the judge checks the balance. The balance has to be equal or greater to the amount of the transaction.

In the case of scheduled transactions, the judge will not check the balance, as the player could have planned to make other agreements with other players between the round where the current transaction is being validated and the round where the payment deadline is set.

That means that a scheduled, or delayed transaction, is not safe by itself, as the judging party cannot guarantee that the payment will be made. Mitigation/punishment in case of lack of payment will be described in section 3.1.

Additional virtual round for transactions validation

It should be noted that, if *deadlines* had been defined so that we had *up to an including* the deadline round to fulfill the transaction, and as delayed transactions are validated at the beginning of rounds, a *virtual* round, that is to say a non playable round, would have been needed at the end of the game. Its purpose would have been to validate the delayed transactions whose deadline are the last (playable) round.

But in our case, as the *deadline* is *up to and excluding* it, and it has to be $\leq$ *last round* (see section 3.1), such an additional virtual round is not needed.

Casting votes aka *voting transactions*

Casting votes is done (by the players), when asked by the judging party, by calling a special method of the judging party. This judging party's method, that is only allowed to be called during voting rounds and just after the juding party has called the player to cast vote, will create a special type of transaction: *Voting Transaction*.

A voting transaction is an *immediate unidirectional transaction* where the resource type is always *votes*. The *payer* is the player casting the vote and the *payee* is the player whose score will be increased by this vote, ie. the player "receiving" the vote.

Voting transactions are created *under the hood* by the judging party when a vote is casted and players do not have to make an explicit call to create a transaction of this type. Transactions of this type, as previously said (section 2.2) cannot be asked to be instantiated by players.

Note that we also had the choice to make "votes" resource a "voting promises" resource and directly enforcing transferred amounts of this resource by the judging party. We would do so by forcing the *payer* player to vote less and casting automatically the *owed* votes at the next voting round or at some voting round specified in the transaction. However, we finally decided to transform every vote casting into a transaction so that not only we keep of every single action in the game as a transaction, allowing for logging, replay and potential error-recovery, but that also allows more flexible voting promises transactions to be created. Indeed, with our implementation, a player can tell another player "OK, I will for sure vote for you X times before the round r, but I get to decide when". Such transactions, in a real world environment, are very important because it allows to add some preassure to some other player for instance, and not to reveal one's true voting agreement too early, while still having strong voting agreements. We believe suchs advantages are better than the sole advantage of having the judging party do it instead of the player, that would prevent the player from cheating. It is to be also noted that this model allows more easily the implementation of *cancellation transactions* as defined in the future work (chapter 5).

Voting rounds

On voting rounds, the judging party will ask players to vote.

Players should vote immediately, no delay is given. Players should vote according to the rules and they should respect any official vote promise agreement they made via a transaction.

The judging party, for every player's vote, will check that it respects the rules, and will then check that it respects vote promises that have been registered via previous transaction. To summarise, a voting round goes through the following steps, in order:

1. Check for scheduled transaction completeness, as for any round.
2. Ask every player, one by one, to cast one vote.
3. Validate the vote according to the rules.
4. Either accept the vote, of qualify voting player as a cheater if the vote was not valid.

5. Do 3. and 4. again for the next player, until we have gone through all of them.
6. Publicly disclose the new number of votes that every player received and their new score.
7. Go to next round (as per section 2.2)

Cheaters ban and resources owed

If, at the moment a cheater is banned, she owes some resources to another player, the remaining resources of the banned player will be transferred to the one they were owed to, up the owed amount, and up to the remaining balance of this resource on this banned player account.

If she owes resources to *multiple* players. The judging party will distribute the money proportionately, rounded to the closest integer. The exact formula is:

$$part\ of\ the\ remaining\ balance\ you\ get = round\left(\frac{total\ amount\ cheater\ owed\ to\ you}{total\ debt\ of\ the\ cheater}\right)$$

Where *round* is the function that rounds a real number to the closest integer.

Last round's additional bound check

We stated in section 2.2 that a "round" passes everytime a *transaction* is applied. We also defined in section 2.2 that *bidirectional* transaction were composed of two *unidirectional* transactions, thus making *bidirectional* transactions effectively ticking the "rounds clock" twice.

Now let us call MAX the maximum round, that ends the game when reached. This means that when we are at round $MAX - 1$, we need an additional bound check for *bidirectional* transactions, or more simply put, we should not allow *bidirectional* transactions if the round's number is strictly higher than $MAX - 2$.

Indeed, if we were to allow it, it would allow the last playing player to cheat the game in the following way:

1. Create a bidirectional transaction with other player X.
2. Propose to buy a high amount of score for all the money you have left.
3. The other player is likely to accept as it is the last round anyway and everything is more or less decided in terms of score.
4. The first part of the *bidirectional* transaction is applied, you get the score points.
5. Then the maximum round number is reached, the game is over, and you have not paid back the other player!

This, obviously, is a form of cheating and thus will be prevented by not allowing bidirectional transactions if the current round number is $> MAX - 2$

3.2 Miscellaneous security measures

Python-related security measures

Objects Tampering

As *final* objects do not really exist in Python, we have to make sure critical objects are not accessible by the players.

For instance, the judging party object is not passed to the players, only pointers to its methods. Thus, players can still call methods of the judging party but cannot override any attribute of the judging party's instance, like resources balances, for instance.

Another example is transactions objects. If transactions are instantiated by the player and passed to the judge. A player could try to change the transaction object's attributes between it is validated and applied, thus making the transaction applied without having been checked on the attributes values it holds when it is applied.

In order to mitigate this, at first we thought we would make a copy of the transaction passed by the player. The player thus does not have a pointer to the object we are going to use anymore. We could then validate and apply the transaction without risks of tampering. The same process would be used when passing transactions as a parameter at the step of "player agreement" check (where we check the player agrees with the currently being validated transaction).

Unfortunately, in Python, classes are also changeable. If the player gets an object of class A, he would in fact be able to change the A class' methods, assigning them to another player-made function, for instance. While one would think such changes would be local to the module, they are applied process-wide. Thus, passing the Transaction object to the player is not possible at all unless we have a run-time enforcer that sets all methods back to the original functions before critical operations on Transactions. But this means additional code complexity, and also, players do not really need access to full-blown Transaction object in the end. They just need to be able to set transactions with *amounts*, *delays* and *target players*. As a consequence, and following the *principle of the least privilege* (section 4.1) we finally decided to go for the other solution, that is to say not to pass to the player the Transaction object. Instead, we group the necessary information for the player to be able to agree or not upon a transaction using the built-in *dictionary* type for easy access from the player, and we do not reuse this object nor make any relationship from or to it from the "safe area" of the program (the judging party).

Threading and Process Memory Tampering

Another threat is that the player would spawn threads that would then be used to launch itself again while it is not its turn or trying to bruteforce CPU preemption to try to make a player's turn infinite.

At the same time, some tools are also available in the Python library that allows code memory inspection of the current running process and could lead to forcing the program into jumping to an arbitrary line of code.

After careful thought about these two issues, we decided that players' code should not need any *import* statement in order to function properly. Thus, we will forbid *import* statements and at the same time access to any native library or python built-in libraries to the players' code. A pre-processing could be applied on players' code that would either automatically remove *import* statements or simply refuse the code's submission if it finds an *import* statement.

Standard Python, without using native extensions (additional ones or Python standard library's ones) does not allow tampering with memory nor threading and thus we wil avoid those two issues.

Implementation-specific security measures (not Python-specific)

As we decided to run all code in the same process, the players' code could run forever, never giving back the CPU to the game's main control code. In order to avoid that, we simply spawn a thread and move the players' code execution to this thread just before every player turn. After a given timeout (the maximum time a player is allowed to play as defined by the rules), the main process will simply kill the running thread if it is still running, and also consider the player as a cheater.

3.3 Implementation of the game/game execution flow

3.4 Global overview of the game's organization

Figure 1 presents a global overview of the game. The game first initializes (cf. *Game Initialization* (section 6) for more details). The game then calls the judging party and the latter is responsible for asking the players to actually play.

3.5 Rounds

Non-voting round

Figure 2 is the sequence diagram of a round. The pseudo-code of a non-voting round is located in the appendix (section 6)

Voting round

Figure 3 presents the sequence diagram of a voting round. The pseudo-code of a voting round is presented in the appendix (section 6).

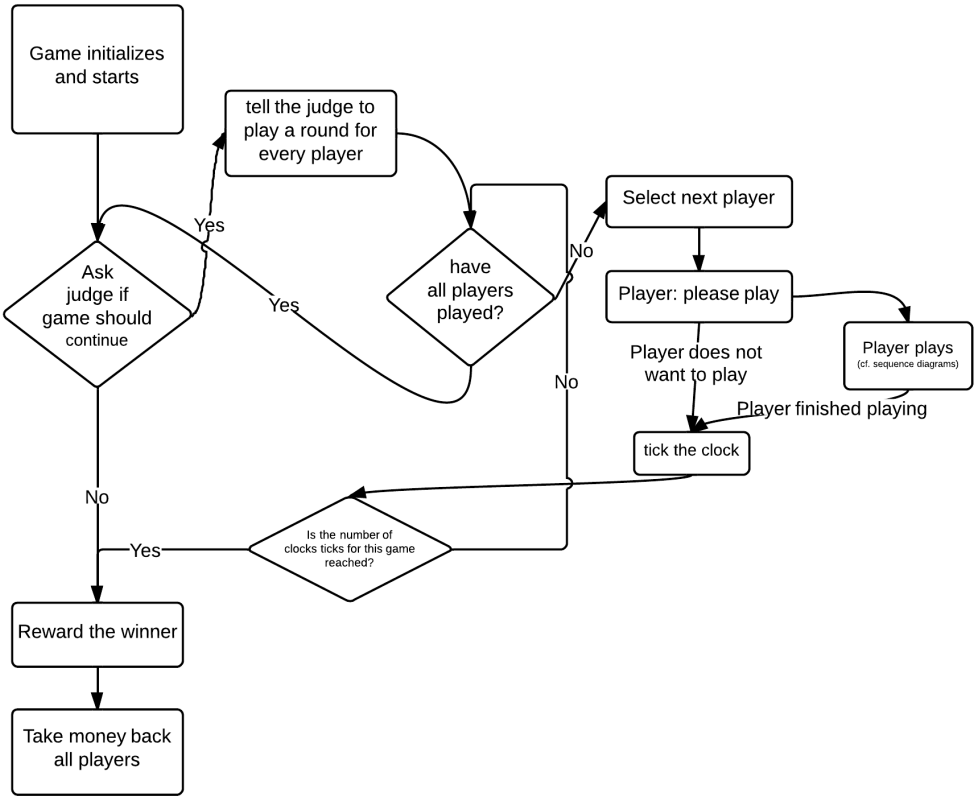

Figure 1: Global Flowchart

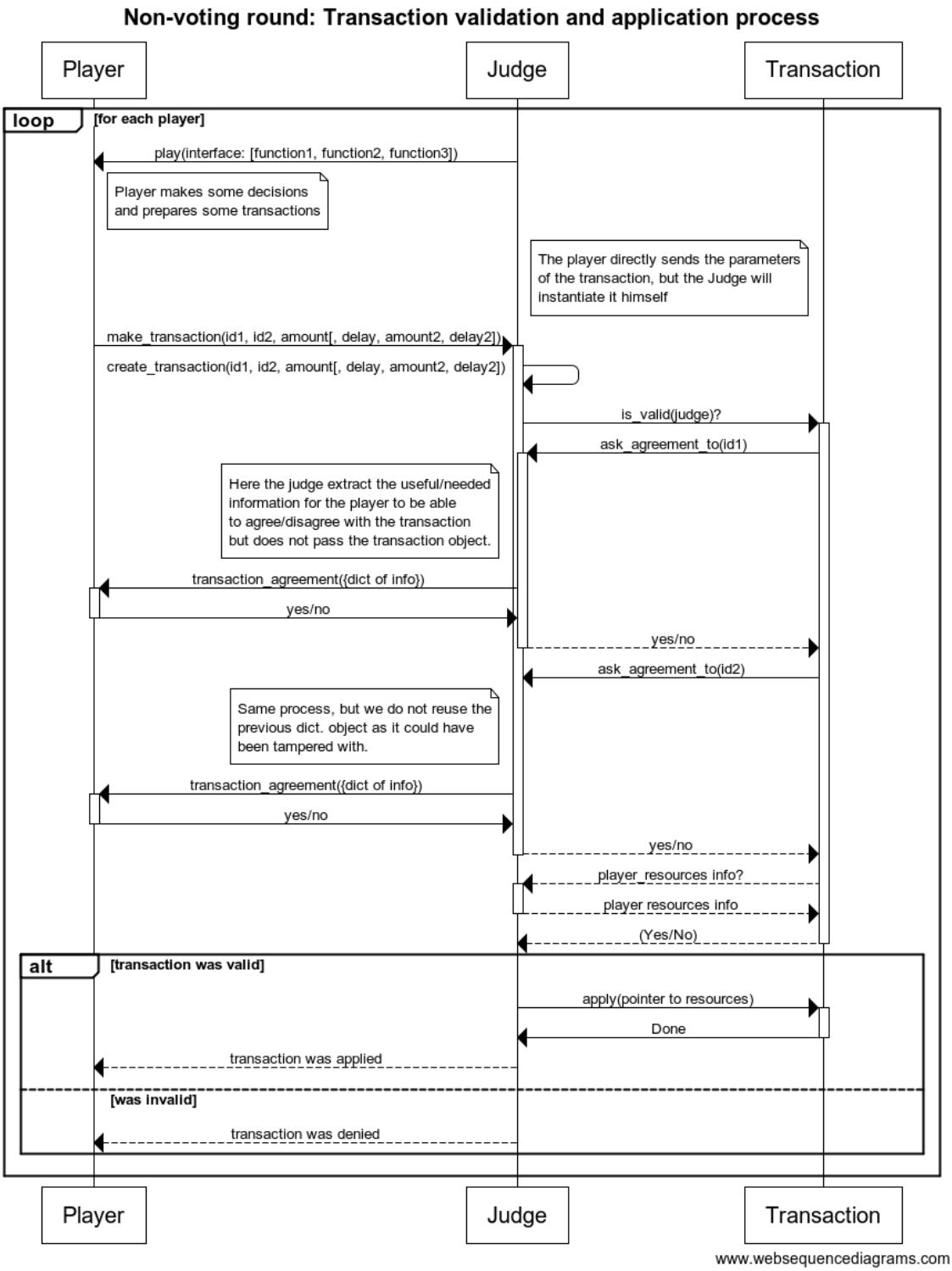

Figure 2: Non-voting Round Sequence Diagram

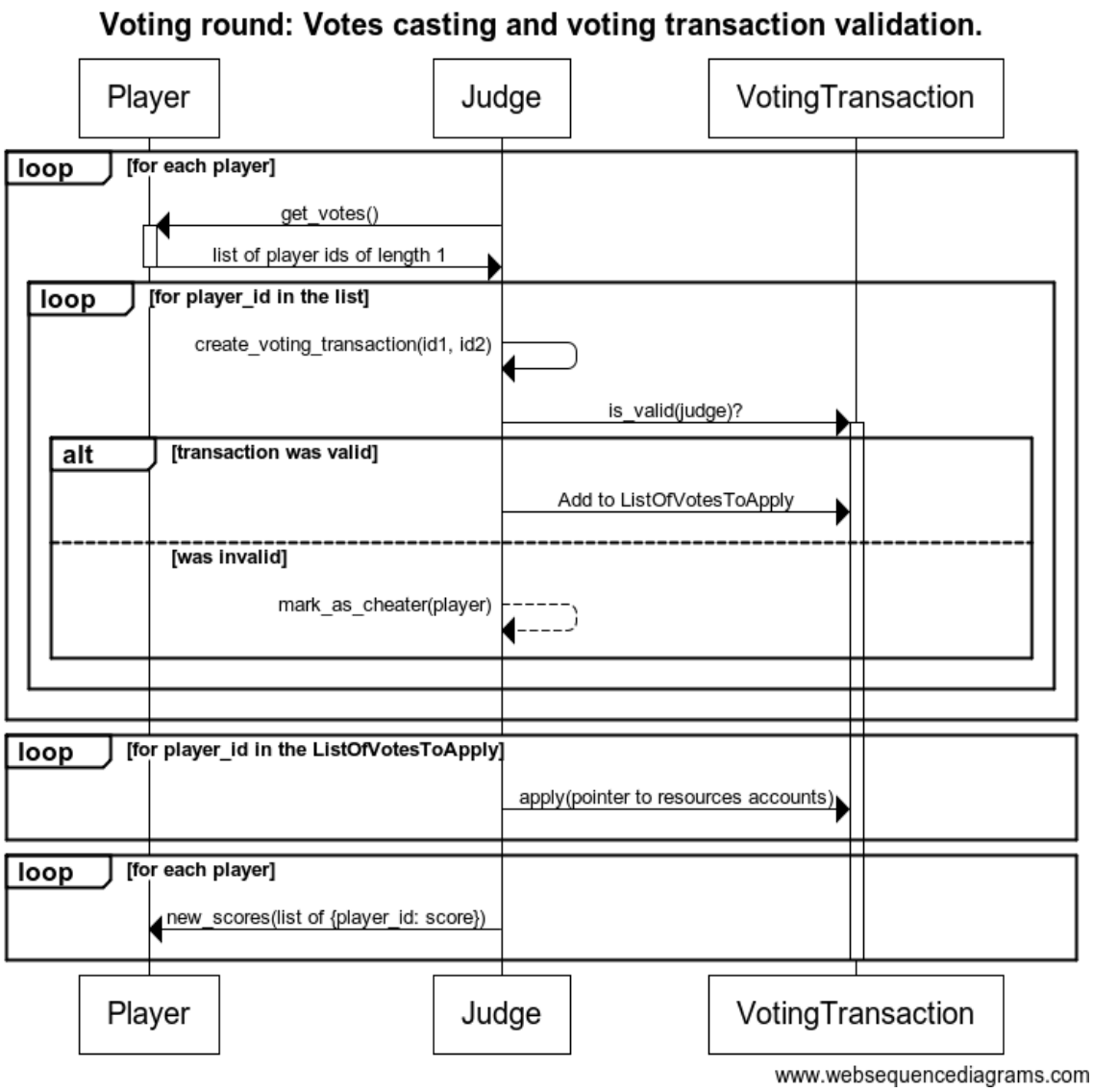

Figure 3: Voting Round Sequence Diagram

4 Coding Guidelines Comparison

4.1 Comparison methodology

We will compare against the guidelines in the following way: we will first explain which themes of the guidelines apply in our case and which do not, and justify why so. Then, for the guidelines that do apply, we will make our best to define how much we respect the guidelines or not in our implementation, justified based on the security features/mechanisms that we implemented.

The complete list of guidelines themes [2] is the following:

- Secure the Weakest Link
- Practice Defense in Depth
- Fail Securely
- Follow the Principle of Least Privilege
- Compartmentalize
- Keep It Simple
- Promote Privacy
- Remember That Hiding Secrets Is Hard
- Be Reluctant to Trust
- Use Your Community Resources

Application of the guidelines to our implementation

We believe the following guidelines do not apply (or not completely) in our case:

- "KISS": The KISS chapter of the guidelines is separated between how the *program itself* should be simple and how it should be simple *for the end user* (human being). The *end user* part of it is not really relevant for us as we do not really have *users* but more *potential attackers* that are the players. We will thus only review the part about the *program itself*.
- "Promote Privacy": This chapter is directly related to *end users* and even more to *human beings*, it does not apply to our implementaiton that has Python AI code as *end user*.
- "Remember That Hiding Secrets Is Hard": Our program is both OpenSource and not compiled. Moreover, we do not use any secret-based technique like serial or private key or seed. Moreover, this guideline is mostly about commercial-related activities, protecting data that belongs to someone, etc. which is not really our case.

- "Be Reluctant To Trust" & "Use Your Community Resources": Those two chapters of the guidelines are entirely focused on off-the-shell components and libraries. They invite not to trust blindly and explain trust is transitive. They also deal with the fact *widely used* software is *less likely* to be buggy but without being a certitude. We do not use any external library to implement security features, thus there is not much to review about those two.

The other guidelines apply and we will directly talk about how well they are respected in details.

Secure the Weakest Link

Secure the Weakest Link theme is, according to the authors of the guidelines [2] about always targeting the implementation of security features to protect the currently weakest link in the overall security chain. The reasonning behind this statement is that attackers will always use the path of the least resistance when attacking a system. For instance they will not try to break your encryption, they will just try to directly get access to a part of the system where the information is stored unencrypted, or get access to the decryption key using social engineering for instance.

In our case, in the early stage of the implementation, the weakest link could have been said to be the "Transaction" component/class. While the transaction object did do some checks by itself, it was prone to be tampered with (this is a Python-specific security point (section 3.2)). When the time came to review the current state of security in the player-to-player trade chain, we could have tried to add more sophisticated checks to the judging party, to be able to deal with Transaction objects that would have been tampered with. But instead, the first thing we did, was to actually sort-of *secure* the Transactions objects. The way we did it was by keeping them in a place where the players never actually accesses them. Then, instead of passing the players a transaction object, pass them a built-in type (*dict*) that is isolated from the rest of the security chain. The isolation is done so that we do not reuse the object afterwards and it has no pointers to any part of the "safe area" of the program nor any other part of the judging party has any link to this object after it is passed to the player.

Practice Defense in Depth

The summary of the *defense in depth* in the book is that having two consecutive layers of security, which work different ways, will hopefully allow you to block attackers that made their way through one of them, with the remaining one.

The analogy is made with a bank. An armed guard should normally be enough to prevent someone from holding up a bank. But, some attackers might be numerous enough or armed enough so that the guard cannot stop them. In this event, having a second layer of security that will, for instance, prevent more than a given amout of money to be taken away, because there are additional security requirements to get more than this amount, will likely help. Indeed, it will first stop some attackers completely, if they are not tooled enough, for instance to break into the vault. Or in other cases it will slow them down significantly and thus, potentially give enough time for the first level of security breach to be detected and someone to act (in the case of the bank, for the police to come).

In our case, for instance, this has been applied to transaction validation.

Indeed, the current user is the only one who is supposed to be able to submit transactions, because it is when its methods get run by the main loop. But if a player manages to get some CPU time outside of its own turn (by managing to spawn a thread or using a Timer for instance), the judging party will still check for the id of the player submitting the request, to be the current player id. The cheater will thus need to also guess the id of the current player.

In addition to this, even if the cheater bruteforces the system and finds the right current player id, the judging party, before validating a transaction that involves a given player, will directly ask the player (direct access, no possible impersonation/man-in-the-middle) if she is indeed OK with the transaction (even if she did submit it herself).

And, assuming the cheater is smart enough to only perform transaction that do not involve the current player, the judging party will still ask the current player whether he is OK that the given transaction is applied on her "quota of transactions" for the current round.

The "quota" itself is another "layer" of security. It will prevent a potential attacker that would have found a breach, to exploit it too often, thus reducing the overall impact and slowing down the attack.

And last, players are also authenticated using a password. Thus, once the cheater has figured out what player *id* is the one of the current player, he then also has to guess its password.

Fail Securily

The idea behind "fail securely" is not much about something *to do* but more about things *not to do*. The basic idea is: make sure that when you end up in a failure state, you do not fall back to an unsecure mode. Examples are given where, for instance when a client and a server software could not find any common authentication protocol, the client would simply download a new authentication protocol from the server. Then, a rogue server could simply refuse all the protocols a client would currently be compatible with and force the client into downloading a new authentication protocol that is in fact a rogue one and will authenticate this server as another secure server.

In our case, such case of unsecure failure could have happened if, for instance, we were to give *reasons* for transactions to be refused.

More precisely, imagine if, when the "buyer" does not have the sufficient funds, we were to return to the "seller" saying "the transaction was denied because of a too low balance". We do not give the balance information here, so it seems OK, does not it? In fact it is not. Because one could simply guess or even bisect (in case the number of tries is reduced) the balance of the other player by submitting transactions of decreasing amounts (starting at a very high amount), until the transaction validates or the *reason* changes.

Having the balance information of another player is both an exploit of the system and a sort of cheating. It is an exploit because it is not supposed to happen, and you would be the only one with this "power", which makes the game unfair. But it also is a sort of cheating as it enables the attacker to trade with more sophisticated information, maximizing its profit, and avoiding potentially payment defaults by not selling to players that have a too low balance.

Follow the Principle of Least Privilege

The "least privilege principle" is a reknown paradigm in security, not only in computer science: Even if you have the highest level of trust in a person, if you give him or her

more privileges than he/she needs, you are putting yourself at higher risks than if you did not. Indeed, even if you are giving the key of your house, to feed the pets while you are in vacation, to your life-long best friend, the probability that something unwanted happens in your house is not zero.

That could even be indirectly your friend's fault. He could loose the keys. And if you had isolated your pets into the garage and only given the garage's key to your friend. Then nobody could enter the house (except the garage). We will by the way see that the *compartmentalize* guideline presented later on makes it easier to apply this principle.

In our case, the principle can be applied in many different places.

The first place is for instance the player resources balance. As the player has the starting amount of resources passed to him at the beginning of the game, the player does not need to have any additional information about the balance. We could have had an access control system in order to allow users to read this information but not write it, but that would have been against the principle of the least privilege: here, the players do not need read access to this information, so we do not give it to them.

Another good example, that follows the analogy of the house, is the ability to call the judging party. One could have thought of passing the judging party's object to the players at the beginning. They could then be able to talk to him as they have a reference to him. But this is much more privileges than necessary. What do the players need? They need access to specific methods of the judging party's object. Then, we only give them access to those specific methods: we export the methods pointers and give these to the player. So the player has access to the method, not the entire object, the same way your friend should only have access to the pet area and not the entire house.

The same principle is applied when asking for agreement over the transactions: what do players need in the end? They do not need a Transaction object, they simply need the data that the object is carrying: amounts, delays, type of transaction... So we generate a dictionary that contains this data for easy access and the player is only given this.

Compartmentalize

Compartmentalize is described in the guidelines [2] as a way to limit the amount of damage that can be dealt to a system when an attacker breaks in. This is in some points similar to defense in depth (section 4.1). We will also see in the next guideline (KISS) that it can be in conflict with it sometimes.

Take the analogy of the pet-sitter previously described again: so you have decided that you will not give more privilege than necessary to the pet-sitter. Thus you want to give him or her only access to an area where your pets are confined. Even if your pets are in the garage, if you do not have a separate entrance and lock for the garage, the pet-sitter will still need to go through the house in order to get there, thus making it impossible to give him or her only the least possible privileges.

The basic idea here is to try to break the program down into the smallest viable independant parts. Once this is done, it is easier to only give permission to access the parts that should be accessed and not to the parts that are not supposed to be accessed.

Compartmentalization is only useful if you have multiple roles, though. Indeed, if everyone always needs access to all compartments anyway, then this does not change much.

This guideline thus does not apply much to our implementation. Indeed the only role we have (for external entities) is the one of "player". All players need the same access and there is not much compartmentalization that can be done.

We could, though, maybe consider that for instance, we could have compartmentalized the judging party into several different entities:

- A transaction validator
- A player manager

Those two parts (except some parts of the transaction validation) are basically handled, or accessible and directly manipulated, by the judging party. But the fact that Python allows all sorts of manipulation to objects you have reference to would have made it quite hard anyway to have those two entities communicate with each other synchroneously without providing, to an attacker that would have got access to one of them, access to the other one.

Keep It Simple, Stupid! (KISS)

The KISS principle is reknown outside of security too. The KISS principle is more or less composed in some parts, of tautology: if you increase the complexity of your program, you increase the risk of bugs. This is pretty obvious, as one could argue that bugs have a probability to appear for every line of code.

The guidelines explain that in terms of security, one of the ways the KISS principle can manifest itself is into concentrating security-critical operations in a small number of *choke points*. This will create a small, easily controlled interface where everything has to go through. The analogy is performed with a stadium: the more entrances there are, the more complicated it gets to perform high quality control on absolutely everyone.

The balance between KISS and compartmentalized/defense in depth is that we should break things down into smaller chunks and/or provide several layers of security up to a point where the system still remains simple enough. "Simple enough" being the hard part to define and unfortunately there are no secret general recipes for that.

In our implementation the choke point can indeed be considered as the judging party: by having a centralized interface through which every player has to go in order to communicate with another player, we avoid the need to control several channels of communication with their own specificities and/or security implications each.

The second part of the KISS principle as described in the guidelines explains how *end users* (human beings) should have a simple enough secure system because they will not deal with security themselves. We are not really concerned about this part though as we do not really have any users but we could argue that we have only attackers: indeed, the goal of the players is to win the game and, cheating is one of the ways to achieve that, thus, one of the actions players should try to perform is cheating.

5 Future work: Game complexification

The following parts and/or rules of the game and/or previous decisions could be changed in order to make the game a little bit more complex but a little bit more realistic in some way:

- When a cheater is banned, do distribution of the cheater's remaining resources so that we try to minimize the amount of debts remaining open or so that we minimize the dissatisfaction of the loaners.

- Allowing *cancellation transactions* (together with a *refund*) for *delayed* transactions. E.g.: a player finally does not want to give that many votes to another player, so she asks for cancellation and proposes a given amount as a refund for the cancelling of the "contract". As any other transaction, both players would need to agree. The judging party would then simply *discard* the remaining open delayed transaction upon applying the *cancellation transaction*.
- Give players an interface to declare the tradable resources they have to the judging party. It would allow to model real life trade where businesses might have exclusive resources that they are alone to possess, compared to everyone trading the same resources.
- Introducing loyalty and trust "resources".

6 Appendix

Game initialization

```
Game::init
    clock = initialize_clock()
    judge = initialize_judge()
    decide_of_player_ids()
    players_starting_resources =   allocate starting resources
                                        storage structure for every
                                        player id
    players = instantiate every player with a player id
                   and a copy of its starting resources data
                   and a copy of its allowed "interface" (set of functions)
    judge.setPlayers(players)
```

Pseudo-code of a non-voting round

```
Game::play
    while judge.play_round():
        pass

Judge::play_round
    if clock.is_voting_round():
        return player_voting_round()

    for pid in game.players_ids:
        p = players[pid]
        try:
            current_pid = pid
            p.play_round()
        except PlayerBannedException as e:
            game.loser = e.player
        clock.tick()
        if clock.is_over():
```

```
            return False
        return True
```

Pseudo-code of a voting round

```
Judge::play_voting_round
    votes_to_be_applied = list()
    try:
        for pid in game.players_ids:
            current_pid = pid
            p = players[pid]
            player_votes_list = p.please_vote()
            for player_id in player_votes_list:
                t = create_voting_transaction(current_pid, player_id)
                if not t.is_valid():
                    raise PlayerBannedException(
                        "Player voted incorrectly.",
                        player=current_pid
                    )
                else:
                    votes_to_be_applied.append(t)
    except PlayerBannedException as e:
        game.loser = e.player
    for voting_transaction in votes_to_be_applied:
        voting_transaction.apply(players_scores_accounts)
    clock.tick()
    if clock.is_over():
        return False
    return True
```

Bibliography

[1] Shinobu Kaitani: Downsizing Game, Liar Game

[2] Gary McGraw, John Viega: Building Secure Software: How to Avoid Security Problems the Right Way 2001

[3] "Time of check to time of use" "Checking for Race Conditions in File Accesses" Computing Systems, Vol. 9, No. 2, pp. 131–152. Matt Bishop, Michael Dilger 1996