3
A Model for Stream Based Interactive Storytelling

In this chapter, a Model for Stream Based Interactive Storytelling is presented. This model uses a distributed strategy, removing the need of having to render stories on the client side, which helps in the creation of a Model that can support the process of exhibiting the same story to a mass of users.

This architecture is a proposal of how to handle a mass of users (even thousands or more) in an interactive storytelling system based on video streaming. This thesis will show how this model can support a shared interactive storytelling experience in a broad audience by generating and rendering the stories to be watched on the servers. As these stories will have delay due to the rendering, streaming and transcoding processes (or the broadcasting, in terrestrial television), the model expects stories to have enough time for the users to watch and interact with a set of interaction suggestions while they watch the story. So these interactions are counted as voting sessions that happen in parallel, to be counted after story cycles.

In order to support the mass of users, the model needs to support a big amount of simultaneous interaction. The framework needs to be scalable in order to be able to grow as needed, and provide a server interface that supports multiple interactions is also compatible with multiple platforms.

The model also deals with questions of how to allow all of them to interact, and how to handle these user interactions. In this context, multiple voting strategies are presented, in order to deal with the question of how a mass of votes can be treated and counted.

3.1.
Multiplatform Interactive Storytelling Model

In the proposed model, the main tasks of plot generation and dramatization occur at the server side. Instead of having the dramatization of stories being rendered and controlled on each platform, we propose video streams as responses

to the clients' requests. Figure 10 illustrates the proposed model, where the drama servers send video streams to the clients, which can be several devices in a variety of environments (e.g. windows computers, android smartphones, android tablets, etc.). This allows the same story to be shown consistently for all the chosen environments; furthermore, the most adequate video quality is guaranteed for them. Also, programming efforts will be much lower than the ones that would be necessary to have multiple versions of the software running in multiple platforms.

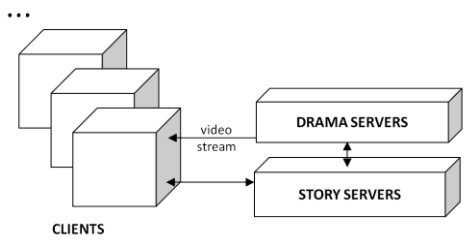

Figure 10 Simplified stream-based interactive storytelling architecture

As far as digital television systems are concerned, there are two distinct types: traditional television systems (terrestrial, satellite, and cable) and IPTV (Internet Protocol Television). Each of them requires a different client structure. For stories in multiuser mode, the same stream is shared through broadcast in a channel. In the case of Brazilian digital TV, for instance, the multiuser mode may be implemented through broadcasting together with a Ginga application, either in Ginga NCL [17], or Ginga-J [18]. The scenario is even more complicated, because smart TVs are using their own software environments.

However, it is important to notice that some peculiarities apply to each environment. Also, the ways of interacting with each one will not be the same. For instance, in a PC, users may use keyboard and mouse. On the other hand, in a Smartphone, the screen and resolution are much smaller and, consequently, the interface cannot occupy the same size and be so full of details.

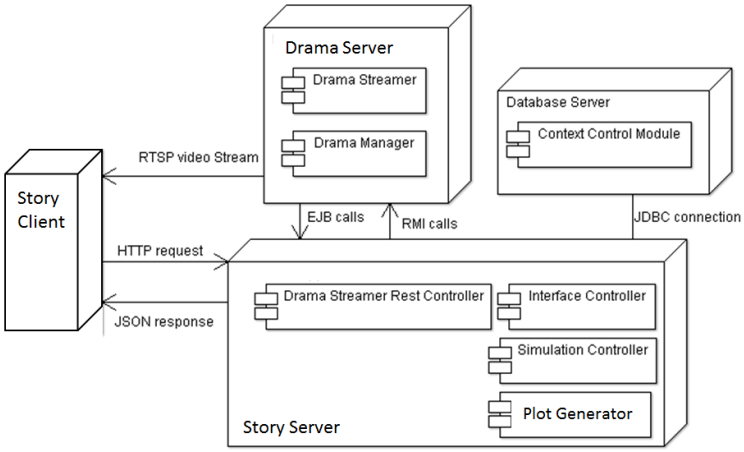

Figure 11 The proposed architecture for the stream-based interactive storytelling system

Therefore, in our model, the choices to interact are defined in an agnostic way, and each client implementation is then responsible for interpreting it in the best way. This means that the interaction possibilities themselves are also served through remote calls, and each client must implement their way to interpret it, that is, choosing the most appropriate way to represent the interaction possibilities on its envinroment. Figure 11 presents a more detailed architecture of the proposed model. Also this figure shows how the system is scalable. The Story Server environment, responsible for controlling story generation and adaptation, can be composed of multiple application servers, thus being able to support bigger loads.

The Story Client is capable of communicating with the Story Server through HTTP calls through a REST [8] interface. This is a much lighter approach than the use of EJB (Enterprise Java Beans) or Web Service calls, like it was done in our previous work [5]. This approach permits an easier creation of multiple client applications, since the HTTP protocol can be used for easy communication and transport of messages between the client and the story server.

The basis of the proposed architecture is defined by a module of the Drama Server called Drama Streamer. This module is responsible for intermediating the story that is being watched by different spectators.

The Drama Streamer module also follows a scalable strategy. There may be multiple instances of it in a given environment. The system uses the Story Server to find all the instances in a given cluster. Each drama streamer instance is capable of rendering one story at a time, since it captures the video of the server.

After some analysis with the possibilities of streaming with Java, we adopted the use of VLCj, a java version of VideoLAN - an open source video player, encoder, and transcoder. It is capable of supporting the media needs of an interactive story, and it was integrated into our system.

In the proposed architecture, the Story Client should connect to the Drama Streamer server using the RTSP protocol. The choice of VLCj (compatible with windows and Linux) for the prototype's PC version of a client was a natural one, since it was also chosen for the server side for encoding the videos.

For watching a story, a HTTP request is sent to a web server in order to discover which available Drama Streamers are free in the server cluster. This method call is made with a plain request, and a JSON [7] object is returned through the network. JSON objects are a standard data layer that simplifies interoperability between different layers of application, and tends to be simpler than other representations (such as XML).

The Plot Generator module is the core of the story creation: it is responsible for creating the list of events and suggestions that unfolds, using goal-inference rules and constraint logic, in a partially ordered planner algorithm. The Simulation Controller is responsible for generating and managing the story and plot generation process by using the Plot Generator module, creating the plot, managing its states and their interaction Suggestions. Suggestions should be consistent and lead the plot towards different outcomes. This process is based on the story contexts that are stored in the Context Control Module, stored in the Database Server. These contexts define the story's logical rules and constraints, characters, relations, and the story's world initial state.

In this new model, it should be possible to have an abstract definition of how to present a history. By doing so, it is possible to render a story in more than one way, which can be used together with multiple profiles of encoding for different platforms for instance.

Previously, the Logtell rendering of drama was bound to only the unity engine. In the proposed model, they very concept of rendering an event should be decoupled. As such, there were some changes done to how things are organized, shown in Figure 12.

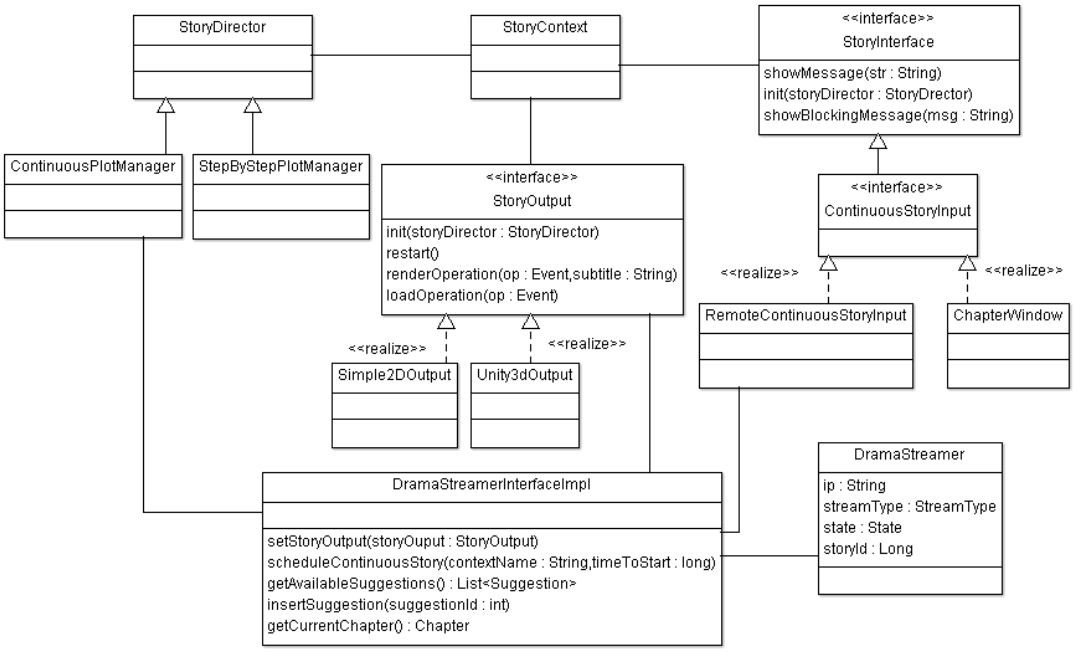

Figure 12 Simplified Class Diagram With new overall model

In this new model, now the Story Output (StoryOuput) is then decoupled from the Plot Manager (which was still present in the current model) and is abstracted as an interface. By doing so, it is now possible to create a 2D output (Simple2DOutput), as a proof-of-concept, other than the current 3D one (Unity3dOutput). By doing this the model is more extensible and adequate to future work, for instance, like using pre-rendered video segments. In the Figure its also possible to see that the plot manager still has its Continuous and Step by Step versions - both are now implementations of the same interface StoryDirector. Everything is bound together by a StoryContext, binding together all aspects of a running story: output, input, and story generation, to be used by the Drama Streamer.

The Interface Controller controls interactions, organizing suggestions from the users. In the figure it is controlled by using the StoryInterface, which manipulates Logtell's story interface controls. It coordinates the simultaneous dramatization and the presentation of suggestions for strong interventions in the various clients. It also controls the time during which users' choices are considered. The proposed architecture is completely different from our previous

model with rich clients [5] in which these clients had to implement the entire logic to render events, keep track of suggestions and votes, request story interactions directly with the servers that are generating them - now everything goes through the Drama Streamer. The control of interactions in our previous work [5] uses EJBs (Enterprise Java Beans) by the clients; now it is controlled by calls redirected from the REST interface.

3.2. A Model for Sharing Massive Interactive Stories

With the explosion of digital communication technology, lots of experiences are called massive, for instance, television, but also Massive Multiplayer Online games, allowing thousands of users to have a shared experience. This works aims to have a model that supports similar capacity.

The most challenging aspect of interactive stories on digital television is how to share desires and interventions from a massive audience on thin clients. We propose a model for sharing massive interactive stories that has two basic forms of user intervention: Local Weak Interaction and Global Strong Interaction.

In the Local Weak Interaction form, the user interferes in the dramatization using local information and plug-ins. For example, the user may ask for another camera point-of-view. This form of intervention heavily depends on the processing capacity of the user's device. Furthermore, this sort of intervention is not shared by other users and has no influence on the plot generation. This was not implemented on the prototype since it is not one of the main purposes of model, but this can be worked on in future research.

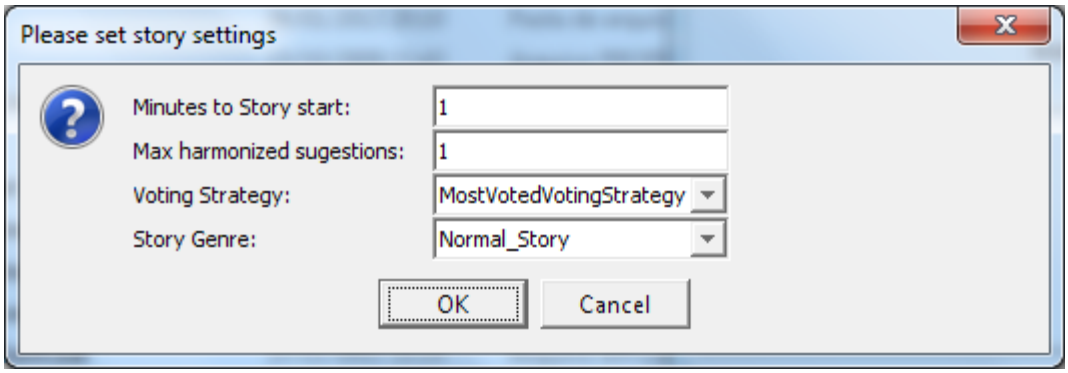

Figure 13 Scheduling a streamed story

When stories are scheduled (Fig. 13), the user (who may or not want to actually watch it - this could be a separate role) can choose some settings on it. These settings are important in this model, since it is adaptable and aims to be flexible on its approach. The settings are:

- **Minutes to start**: amount of time the server should wait before starting a story running.

- **Max Harmonized Suggestions**: the maximum amount of suggestions that can win in a voting session

- **Voting Strategy**: which voting strategy to use to count and choose selected suggestions by users

- **Story Genre**: which story genre/context to show (currently using a fairy tale story - which can have some variations that generate longer or shorter stories)

In the Global Strong Interaction form, the user may change the course of the story. In this form of intervention, we propose the following strategies: Most Voted Suggestion, Weighted Voting, and Harmonized Voting (explained in 3.3).

The way groups are formed to share a story is also an important issue. Off-air broadcast has no flexibility of choice, that is, users should join the story at specific date and time. In other systems of TV, such as IPTV, we have more options. The simplest strategy is to assume that any user is allowed to schedule the start of a story based on a specific context at a certain time. As soon as other users notice a specific story in a list of scheduled stories, they may be tempted to join the group. Users can either have equal rights to intervene in the story or not;

in the latter case, different criteria can be established to assign their rights and privileges. Independent of the TV system (off-air, IPTV, internet TV), methods for the communication among users who share the same story can be adopted. In this case, users can discuss their interventions. An interesting model of incorporating social networks into an interactive storytelling system can be found elsewhere [20].

We consider the question of user interface modalities as an issue of utmost importance for massive interactive storytelling. Another work by the Logtell research group proposes a multimodal, multi-user and adaptive interaction model [21]. The experimental production of the configurable documentary "Golden Age" by the ShapeShifting TV research group [22][23] is a valuable source of information that could be implemented in the interface of our system.

During the tests of the prototype, we have created multiple voting strategies. Also, we implemented a simple interface modality in the prototype, as presented in the Chapter 4. Usability issues from the viewpoint of Human-Computer Interface were not considered in the present work. We made these decisions because the focus of the present thesis is on the client-server architecture.

3.3. Voting Strategies

In our model, multiple voting strategies must be available in order to adapt to different user needs. Different story contexts and audiences may want a different behavior in how choices are made.

In the Most Voted Suggestion strategy, when users ask the system to incorporate a specific suggestion, they are engaging themselves in a simple process of vote counting. The most voted suggestion will be chosen to be incorporated in the next chapter. The Interface Controller organizes the interaction with clients and interacts with the Simulation Controller as if there were a single user. In order to do that, it coordinates the simultaneous dramatization and the presentation of suggestions for global strong interventions in the various clients. Also it controls the time during which users' choices are considered. After computing the most voted suggestion, the Interface Controller checks whether the

number of votes reaches a minimum threshold. If this is the case, the suggestion is sent to the Simulation Controller.

Weighted Voting is a strategy of global strong interaction in which the number of votes can be weighted by the potential of each option to trigger goal-inference rules. In this way, options that generate more interesting situations tend to be chosen.

In the Harmonized Voting strategy, compatible interventions can be combined in the same chapter. In particular, different groups of users may have different options. For instance, the groups can be separated by the characters they decide to support and, then, the planning algorithm (Plot Generator module) tries to combine the choices of all groups.

So, the model is made more complete by having the voting strategy itself as an option, whether it be a user (or moderator) working for a TV channel, for instance, or for the internet itself, depending on what infrastructure the model implementation is being used on, where users may have their own channels for their desired audience. As such, some possible ways to count votes, inspired by voting literature, were created for this model.

3.3.1. User Model and Voter Importance

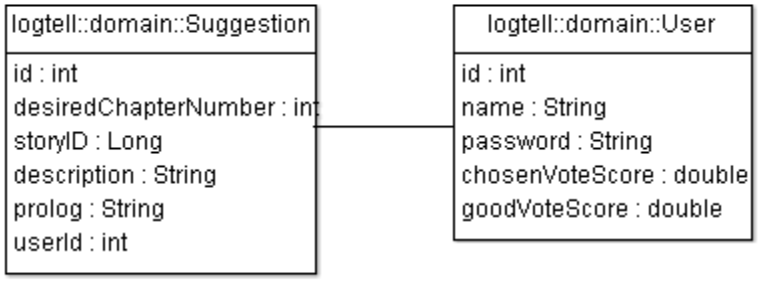

Figure 14 Simple Suggestion / User Model

In order to be able to support the proposed voting strategies, the architecture needs a user model. So, a simple user model (Fig. 14) was created to store a score

for the user, according the ideas presented in [36][37], that is, the notion that users that vote together for the same choice are usually "better" voters.

As such, in our model we have created a link between a suggestion and which user voted for it. Also, we store for the user two kinds of score. One is the "Chosen Vote Score", which gives the score for the times the user voted for something that was in fact accepted for the history. This also takes in consideration harmonized suggestions, where more than one suggestion is accepted.

The second type of score that is kept for each voting user is the "Good Vote Score". Based on the notion that "collective groups choosing a vote" is something good, it can also be a kind of indicator that somehow this user is attuned with the other voters.

To support these new scores, the Continuous Story model of Logtell was changed, so that every time a user is either rewarded or penalized (if the user chooses some suggestion alone, then the "good" and "chosen" scores are penalized). The objective here is to provide some sort of balance and to avoid users abusing the system and keeping the same power forever; which would not be fair if we consider the notion of time, and multiple story sessions.


```
countAndUseSuggestions (suggestions) {
  group all users who voted in each suggestion
  pick suggestions using story's voting strategy
  apply voting harmonization to suggestions
  increase good score for voters who picked winners
  reduce chosen score for voters who picked winners
  increase chosen score for voter who picked losers
together (more than one vote)
  reduce both scores for voters who voted alone
}
```

Figure 15 New continuous flow and rewards

Figure 15 shows the model working. We can see that any user who takes part in a voting session, will have their score affected. This is done in order to keep them "organic" that is, voters who participate will have fluctuations in their voting scores, instead of keeping them static. Note that the model is flexible,

allowing the system to use more than one voting strategy and harmonization setups.

3.3.2. Voting Strategies and Methods

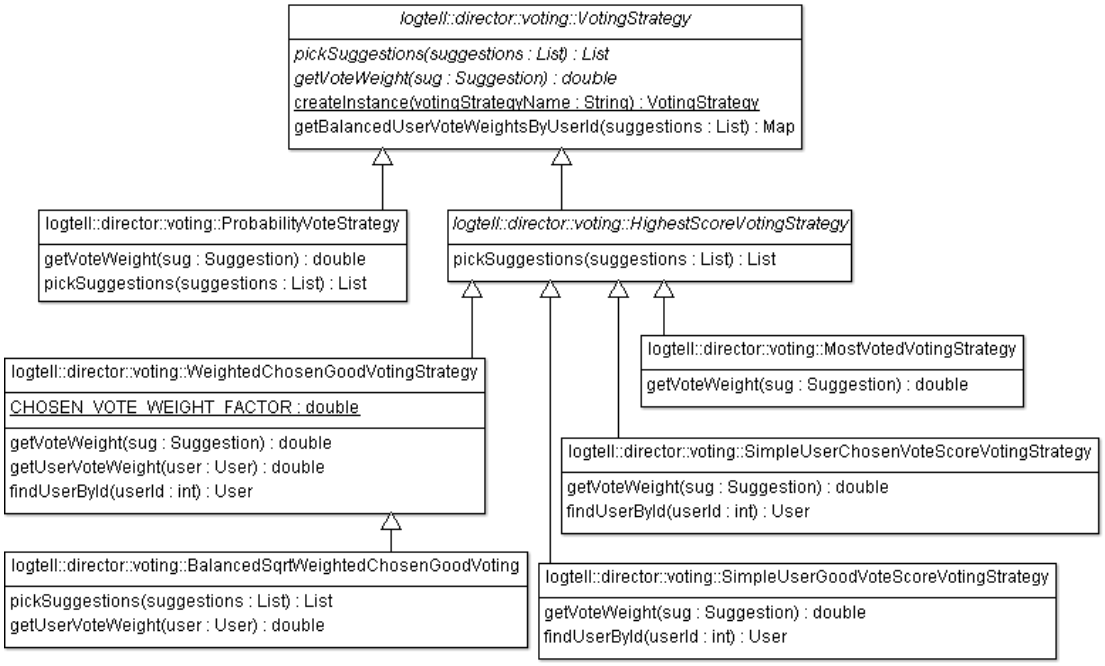

Figure 16 Voting Strategies

In the diagram of Figure 16, it is possible to see how we modeled the voting strategies. In fact, our voting strategies are not voting in the traditional way, since it is actually used as a form of ranking. This makes sense specially because of the Harmonized Voting, explained in section 3.4.

For simplicity, assume that in our model we focused on simple Plurality voting. This is because other forms of voting may require too much interaction from each user, which does not comply with the usual way of watching TV and general philosophy of this work. This means that "the highest voted option wins", regardless if it has reached majority or not, also based on the "least commitment" strategy of the possible interactivity.

In the rest of this section we present the voting strategies of Figure 16 with more details.

- **Voting Strategy**: This is the abstract base voting strategy. It is not an actual implementation, but a base definition. Basically this defines that any voting strategy will receive a list of suggestions, unordered, and then ranks them according to the actual strategy. It also must define a way to calculate the voting weight of any given suggestion.

```
pickSuggestions (suggestions) {
  for each suggestion {
    calculate the vote weight for suggestion
    accumulate total score for each similar suggestion
  }
  sort suggestion accumulated scores
  If there is more than one top score suggestion {
    add a random top score suggestion to result
  }
  add the rest of the ranked suggestions to result
  return ranked suggestions
}
```

Figure 17 Highest Score Voting Strategy

- **Highest Score Voting Strategy** (Figure 17): this is the **abstract** base for the used "Plurality" Voting overall strategy in the model. So its not really a voting strategy, but a template voting strategy. It basically means that suggestions will be grouped by votes and then counted somehow, where the highest score wins. The child strategies implement different voting weights.

```
pickSuggestions (suggestions) {
  for each suggestion {
    calculate the vote weight for suggestion
    accumulate total score for each similar suggestion
  }
  sort suggestion accumulated scores
  If there is more than one top score suggestion {
    add a random top score suggestion to result
  }
  add the rest of the ranked suggestions to result
```

```
  return ranked suggestions
}
calculate vote weight (suggestion) {
  return 1
}
```

Figure 18 Most Voted Voting Strategy

- **Most Voted Voting Strategy** (Figure 18): this is a simpler form of vote count - each individual user vote counts as one vote, that is, weights one. This is equivalent to a simple plurality voting where the most voted one wins (with the addition of choosing randomly when there is a tie).

```
pickSuggestions (suggestions) {
  for each suggestion {
    calculate the vote weight for suggestion
    accumulate total score for each similar suggestion
  }
  sort suggestion accumulated scores
  If there is more than one top score suggestion {
    add a random top score suggestion to result
  }
  add the rest of the ranked suggestions to result
  return ranked suggestions
}
calculate vote weight (suggestion) {
  if the suggestion belongs to a logged user {
    return user chosen vote score
  } else {
    return 1
  }
}
```

Figure 19 Simple User Chosen Vote Strategy

- **Simple User Chosen Vote Score Strategy** (Figure 19): this is a voting strategy that takes into consideration each user's "Chosen Vote" score - that is, instead of using 1 as vote weight, uses the current value of "chosen" score. This

might be preferred for when a user only wants experienced users input, because new users will begin with a low weight.

```
pickSuggestions (suggestions) {
  for each suggestion {
    calculate the vote weight for suggestion
    accumulate total score for each similar suggestion
  }
  sort suggestion accumulated scores
  If there is more than one top score suggestion {
    add a random top score suggestion to result
  }
  add the rest of the ranked suggestions to result
  return ranked suggestions
}
calculate vote weight (suggestion) {
  if the suggestion belongs to a logged user {
    return user good vote score
  } else {
    return 1
  }
}
```

Figure 20 Simple User Good Vote Strategy

- **Simple User Good Vote Score Strategy** (Figure 20): this is similar to the "Chosen" strategy, but is less penalizing for new users. Instead of using the "chosen" score as vote weight, it uses the "Good vote". So, users, that have a "collective" attuned mind will be rewarded with more voting weight, regardless of their "good" scores.

```
pickSuggestions (suggestions) {
  for each suggestion {
    calculate the vote weight for suggestion
    accumulate total score for each similar suggestion
  }
  sort suggestion accumulated scores
  If there is more than one top score suggestion {
```

```
    add a random top score suggestion to result
  }
  add the rest of the ranked suggestions to result
  return ranked suggestions
}
calculate vote weight (suggestion) {
  if the suggestion belongs to a logged user {
    return calculated user vote score for suggestion
  } else {
    return 1
  }
}

calculate user vote weight (user) {
  return (user chosen vote * FACTOR) + user good vote
score
}
```

Figure 21 Weighted Chosen Good Vote Strategy

- **Weighted Chosen Good Voting Strategy** (Figure 21): Uses a weighted formula of the user's vote weight itself. Considers the "Chosen" vote 50% more than the "Good" vote, in order to pursue a more balanced representation of the users' decisions over history.

```
pickSuggestions (suggestions) {
  for each suggestion {
    add a random suggestion to result
    remove similar suggestions from set
  }
  return result
}
```

Figure 22 Probability Voting

- **Probability Voting** (Figure 22): This is a voting method that takes into consideration that even with many votes, not necessarily an option should always be chosen. This makes more sense when there are many users watching a story, specially because votes can be manipulated, for instance, if lots of users combine a "mass attack". Also, by using probability, it can promote more variation, since less popular / strong votes will always have some chance of winning, while still

giving more chance to the most popular choices. It works by picking randomly from all votes a choice (in which repeated votes for the same suggestion makes them more probable to be chosen, thus representing the most wanted choices), and then after picking a suggestion, removes all "repeated" votes and keeps on repeating the process until all voted choices are ranked according to this probability process.

```
balance user vote weights (suggestions) {
  for each suggestion {
    calculate each suggestion normal user vote weight
  }
  while not balanced {
    calculate top weight
    calculate sum of all weights except the top one
    if (top weight > (sum without top / 2)) {
      for each suggestion {
        suggestion weight = square root(weight)
      }
    } else {
      balanced = true
    }
  }
  for each suggestion {
    add user balanced weights to result
  }
  return result
}
```

Figure 23 Balanced Square Root

- Balanced Square Root Weighted Chosen Good Voting:

This is one of the most important strategies of the model. Instead of fully rewarding users votes regardless of the other voters taking part in the same voting session, it tries to reach a balance. It is an extension of the Weighted Chosen Good Voting Strategy, but when summing up votes, it uses a balanced voting weight for each user vote, instead of the normal weighted vote formula.

In Figure 23 we can see the pseudo code for the balancing part of this voting strategy. Note that this function is part of the abstract voting strategy: in the

future, other voting strategies may use the same balancing strategy, regardless of their original calculated vote weight for each user.

This balance seems important because otherwise some users would always dominate story sessions, since it would be possible for even with a big mass of users to accumulate a big weight, which then could make the experience less rewarding for spectators that want to interact.

Focused on the notion of avoiding Dictators, as mentioned in the Voting Methods literature, this voting strategy looks at the entire body of users and based on their weights, balances the more powerful voters in a way that they can't be dictators and decide for themselves.

In order to reach this objective, this model is inspired by the works of Penrose [40], and also the mathematical notion that if we apply the square root function to all numbers in a set, they tend to keep a reasonable proportion. So, this method is an extension of the "Weighted Chosen Good Voting", but before actually counting users votes, adjusts each user vote weight by applying this method of recalculating each voter weight to be no more than the sum of all other voters' weights, divided by two (that is, the majority of all their weights combined). By doing so, this model can ensure that experienced users have more power, while still making sure that they will never have full control over the story.

3.4. Voting Harmonization

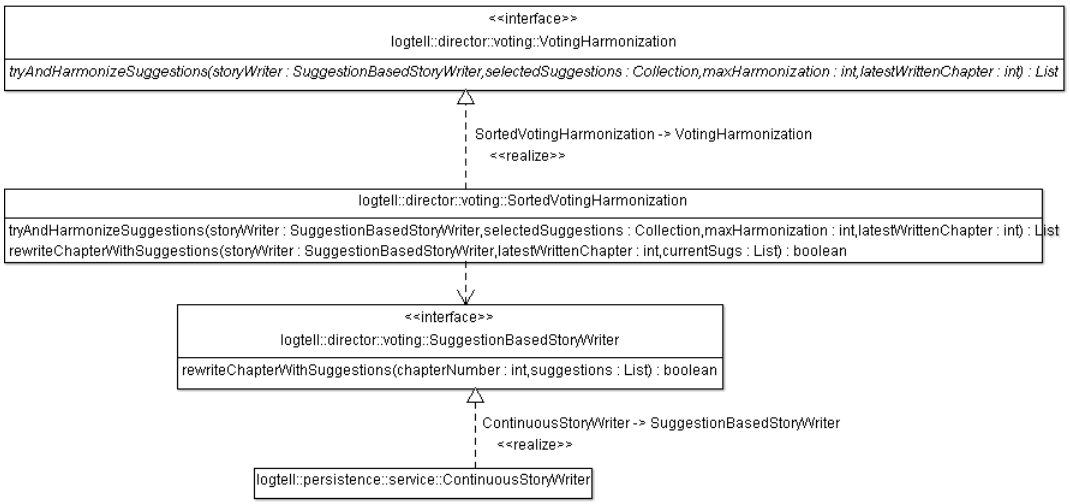

Figure 24 Voting Harmonization

Voting harmonization is the possibility of accepting more than one user interference at the same time. In the diagram of Figure 24, we can see that in our model we only have one way of doing this. Basically, it controls, according to the story preference, how many suggestions should be harmonized. This is done by trying to insert, in the ordered ranking produced by the voting strategy, the winning user choices.

```
tryAndHarmonizeSuggestions(storywriter,            ranked
suggestions, max harmonization, chapter) {
  for  each  suggestion  and    while  used  count  <
   max harmonization {
    try to insert suggestion using story writer
    if (suggestion inserted) {
      add used suggestion to result
    }
  }
  return result
}
```

Figure 25 Sorted Voting Harmonization

Figure 25 shows the pseudo code of the model's solution. The approach is to use the story writer (which is the process responsible for writing the story, accepting suggestions, writing chapters, etc) according to the parameters of the story. This is called a "sorted voting harmonization" because it simply follows the ranked suggestions, according to the voting system used, and tries to insert them in their winning rank. Note that the algorithm checks if the suggestion is accepted and if so, inserting it. Anyway, it must return the used suggestions because they are then used to reward the users that got the winning suggestion(s).

In order to make voting harmonization functional, our model bases itself in the logical possibilities. So, iteratively, each suggestion is inserted and accepted if valid. In this way, the model has the possibility of maximizing user satisfaction, since, depending on the story, multiple possible interactions are compatible.

3.5. Streaming Capabilities

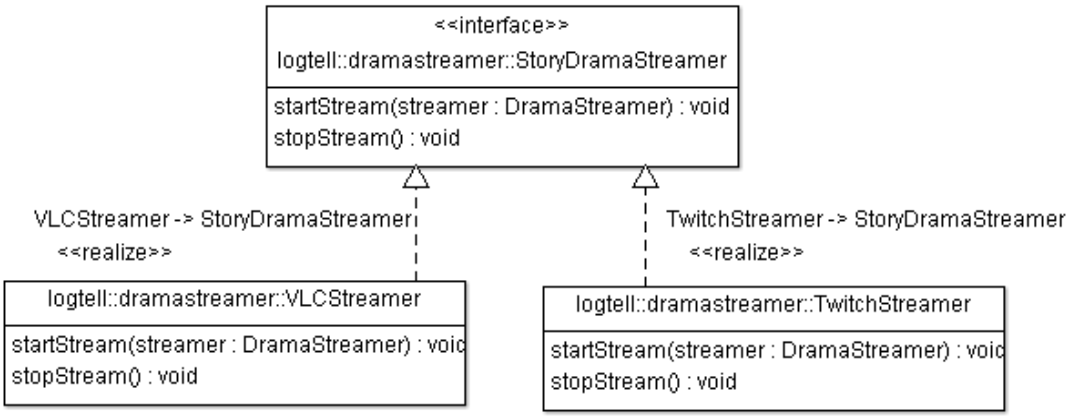

Figure 26 Drama Streamers

As the model is extensible, it should now be possible to serve the story more than one way. So, in the diagram of the Figure 26, we can see that Story Servers, which serve dramatization in streams, can have more than one technique for encoding and streaming, providing support for different types of codecs and formats. This is important for the model, since it allows for easier integration with different real life infrastructures and production level architecture.

If the model is implemented in a real TV channel, probably it will be much better to use specialized hardware for transcoding and sending the actual video streams. So then, it is just a matter of following the model and implementing the connection to the computer where the story is being rendered. Of course, depending on the technique, the clients may have to adapt, since not all clients are capable of watching any kind of stream. That is, the streaming capabilities may use different formats and codecs that will supported for some clients, for instance, a Windows will normally not have TV channels (broadcast over the air), but will be capable of playing MPEG through RTSP streams.