4
Processes and Methods

In this chapter, the prototype implementation of the model is presented, and also relevant points about the implemented version of the architecture are explained. The prototype was created using Logtell's current architecture.

4.1.
Application environment

In the prototype, two main platforms were considered (Windows and Android) and two dramatization outputs (3D rendering using Unity [19] and 2D comics with text). When started, the drama streamer instance sets up its Story Output (by default the *Unity3dOutput*) and starts its *VLCStreamer* instance, responsible for streaming the story video for its clients. Afterwards, the Drama Streamer registers itself on its RMI Registry, which will be called by the JBOSS server of the Story Server environment. Then, the WEB calls, which are received by the Web server, will know to which server the RMI calls should go, when received in its REST interface.

After an available drama streamer is found, the client application should connect to this server. This connection actually occurs in more than one way. The dramatization of the story itself is received by a RTSP video stream, which is available on almost all modern platforms. Other protocols should be possible, but for now this one is being used since it is very portable and commonly supported.

Other than supporting the video stream, Story Clients must also be coded to interact with the servers by using the REST interface. This interface contains all remote methods that need to be invoked in order to watch and control a story.

In this chapter we present the supported methods in the REST interface of the Story Server (Drama Streamer Rest Controller). With only those methods and a video client that can connect the video stream, a new story client can be implemented in any platform.

4.2.
Drama Streamer lifecycle

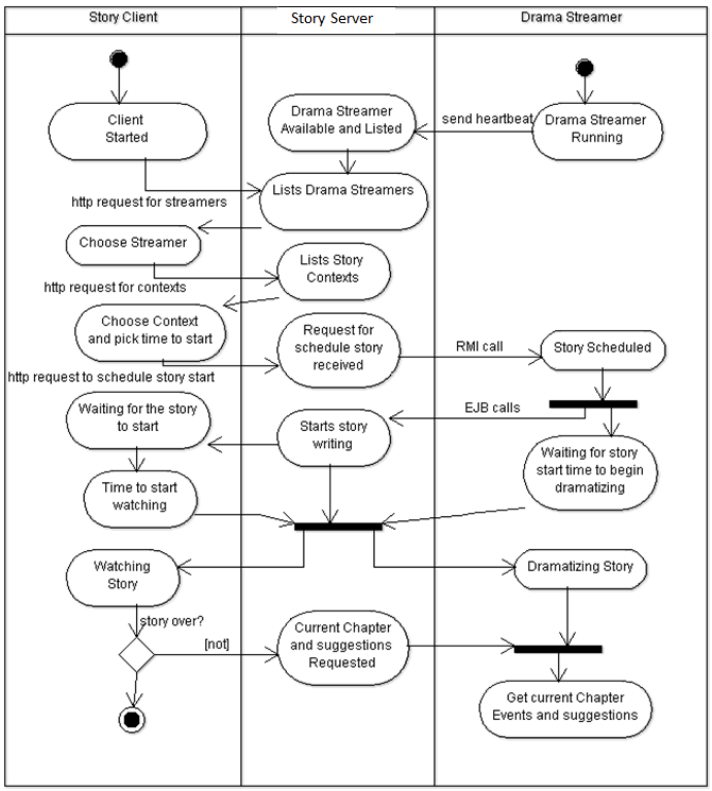

Figure 27 Activity diagram for the process of watching a story

Figure 27 is an activity diagram that shows how the activity of watching a story happens overall. We can see that the Drama Streamer starts and registers itself in the Story Server. Also we can see that all client calls go to the Story Server before going to the Drama Streamer. In this diagram we do not represent the video stream.

Basically, in this architecture, we can see that the client is just responsible for asking the user for preferences and then watching the story. All the story writing is still a responsibility of the Story Server. Any other remote method calls that the client needs are also received by the JBOSS web Server, which then redirects these calls to the Drama Streamer. Then, the Drama Streamer itself may need to make calls back to the JBOSS Application server, since it will ask, for

instance, for the story writing process to start. The story writing process is still the same one used in our previous work [5], which can be accessed for further details.

In the proposed architecture there is a constant heartbeat being sent from the Drama Streamer to the JBOSS Server. This is a crucial aspect, because this is how the architecture will know which Drama Streamers are running, where, and what are their addresses. With this type of information, clients can know where to receive the RTSP stream of story, and the JBOSS server itself can do RMI calls to access specific Drama Streamer story context information and dramatization.

Client applications also connect to the drama streamer to reach the interaction options and chapters information. These calls are made by sending HTTP requests, receiving JSON object representations of the chapters, events, and available suggestions. By indirectly sending messages to the drama streamer, the interaction requests are also sent to the Story Server.

When we use the REST layer, the calls can be done by a simple HTTP request, instead of a more complicate access through Enterprise Java Beans calls. This approach facilitates the implementation and portability of the system to multiple platforms. In this case, the Jersey library was used on the client side, although even if it did not exist, it was just a matter of doing HTTP GET method requests and de-serializing the response in the form of JSON objects. This is a practice that can even be done natively on some languages, like Javascript.

In our prototype there is a delay between the rendering, transcoding, and streaming of the stories. This limitation should be smaller in an actual deployment environment in production use, with dedicated stations, by using better hardware and multiple nodes. Anyway, this should not be a big issue, since interactions in our system are mainly focused on the subsequent chapters – that is, while the user watches events of a given chapter, s/he is given the chance to suggest interactions for the next chapters. Also, in any TV broadcast process, it is inevitable that there is some delay, especially in the case of off-air television.

Figure 28 different story client menus (Android and Windows)

Figure 28 shows two different start menus (on different operational systems and devices) representing the same interface objects and the same story controlled by a common server. These menus reflect different implementations of story clients. One implementation is an Android application, running on a tablet, and the other is a windows application running on a laptop. Both applications are able to access the Story Server through the REST interface and to receive a video stream from the Drama Streamer Server. In the Windows application, VLC is used, but only as an embedded video player. In the android client, the native video player code was used, what shows how flexible the architecture is.

4.3. Prototype Clients

In this section we show how the prototype implements the new "thin" clients, which follow the model. Moreover, we show how the model demonstrates to be portable and multiplatform as intended.

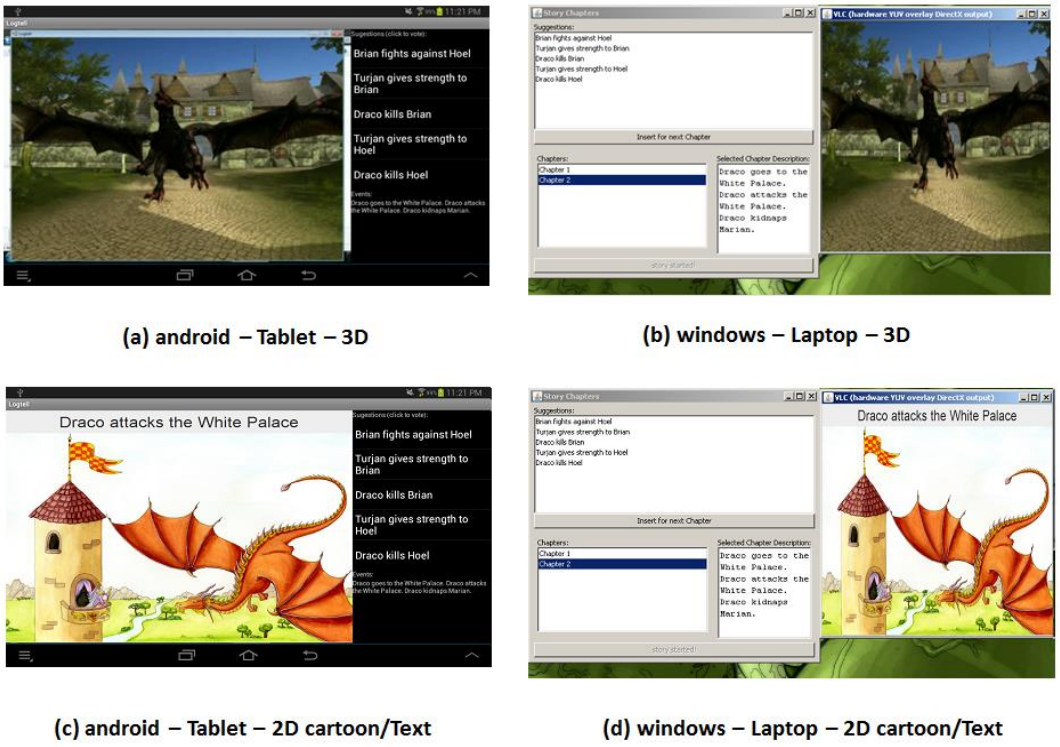

Figure 29 Multiple story clients

In Figure 29, we can see the same story running in two different applications, which are on different devices and operational systems. The difference between the two applications is the dramatization format, that is: one is a 3D rendering and the other one is a 2D cartoon with text. The last format has no animation. Several other formats can be easily implemented in the system, what is another evidence of how flexible the architecture is. Also there would be the possibility of having different forms to interact, such as using icons or voice input. As a design option, the proposed architecture cannot broadcast different dramatization formats simultaneously. Since these applications are running in different platforms, the interfaces can and should be different, because they have different screen resolutions and interaction devices. For instance, in the Android application, the chapter description section is simplified - in (a) and (b) parts of the Figure, the system shows only the list of events in the current chapter; while in (c) and (d), previous chapters' events can be seen at any time. This is only a simple adaptation, but future work can adapt different clients to show, for example, icons and other visual components more adequate to target platform.

Also, the system should be able to be expanded to support different forms of stream. As mentioned in Chapter 2, Twitch is a streaming platform that supports thousands of simultaneous spectators watching the same video stream and interacting with each other through chat. So, other than the main VLC Streamer implementation, a partial implementation using Twitch's network was made. One advantage is that this uses a network specialized in streaming, that can support thousands of simultaneous users, even as a free service.

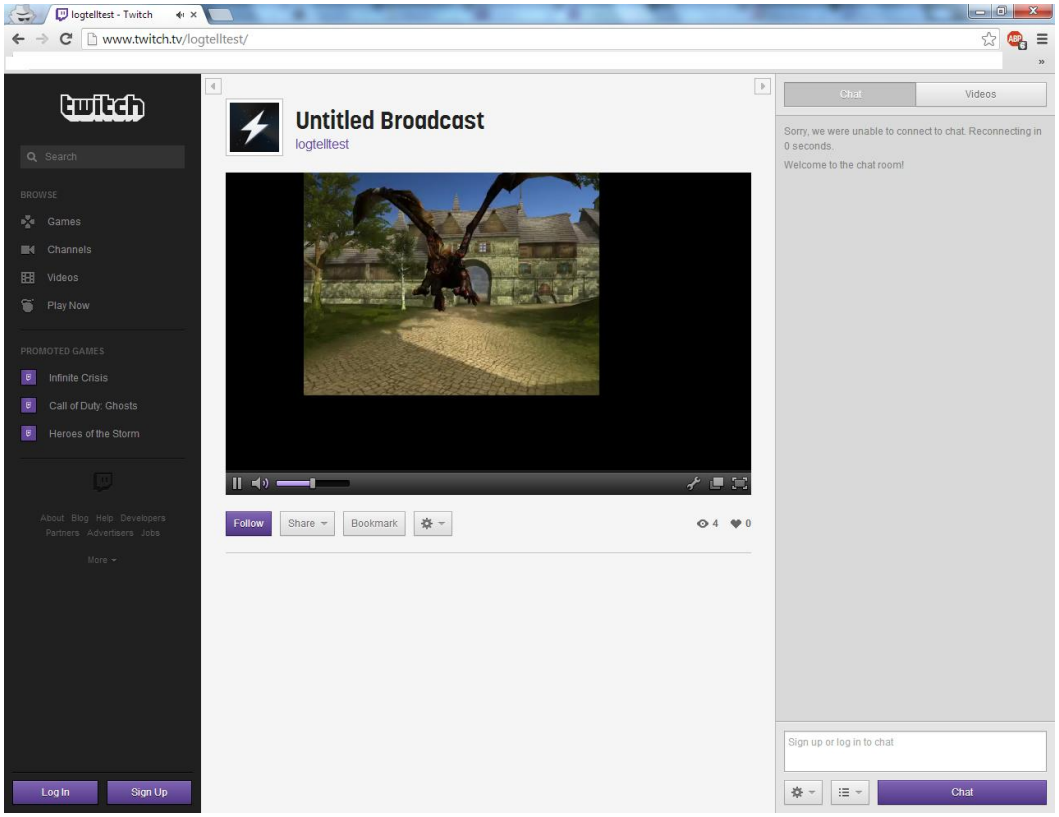

Figure 30 Logtell Twitch Streaming

Figure 30 shows Logtell being broadcast in Twitch servers. By using this service, it shows how the model can be adapted to multiple environments. This was made by adapting an open source streaming solution that supports Twitch, "Open Broadcaster Software" into our prototype, creating TwitchStreamer, implementing the StoryDramaStreamer.

Future work can be done to implement natural language processing in this mode, which would enable some users to watch stories using simply the browser. A partial implementation using the IRC protocol was done, which reads the

messages from the stream channel and interprets them as votes, thus making the stories available to any client of the Twitch platform (HTML5 / Flash and Android clients, for instance).

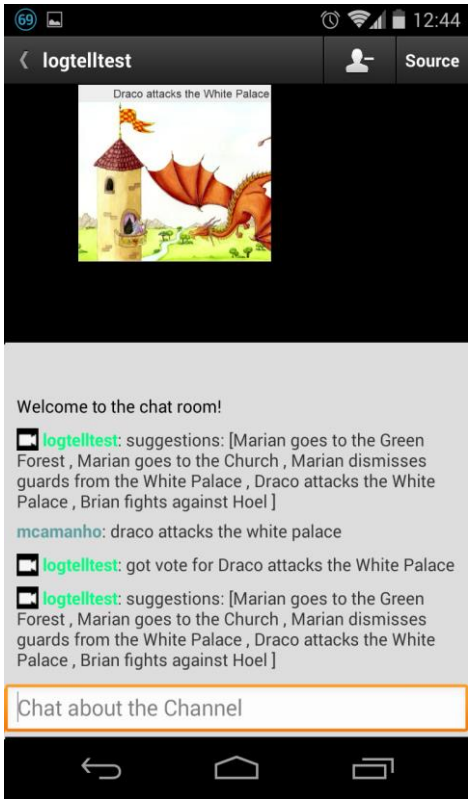

Figure 31 Twitch client using Logtell

Figure 31 shows the example Twitch based implementation, running in a Twitch normal Android client (that is, a Twitch tool for watching any of their video streams). Despite some delay, the use of the Twitch infrastructure provides the access to multiple platforms at once, by using the Twitch clients themselves. In the screenshot we can see how a simple implementation of the IRC protocol (supported by twitch) can be used to make its chat function work as the client. This shows the potential of the model presented in this thesis.

4.4. New Server Interface

The supported methods in the REST interface of the Story Server (Drama Streamer Rest Controller) have special java annotations that define them as REST methods to be called over the HTTP interface. They are named accordingly, using the same name the java method has but separated by dashes. The same occurs to their parameters, showing how simple its access is. We decided to present the methods because they help the reader to understand the flexibility of the proposed system.

We describe below the available methods on the REST interface, for each URL format accepted:

• *getDramatizationServers()*

http://[story-server:port]/REST/streamers

Action: Returns the list of available Drama Streamer servers, their address and state (whether they are idle or already showing a story).

• *getAllVotingStrategies ()*

http://[story-server:port]/REST/streamers/get-all-voting-strategies

Action: Returns the list of available Voting Strategies. Since clients do not need to count votes, they only need the list of voting strategies so that the choice of which one to be used in a new story is done.

• *getAllContexts()*

http://[story-server:port]/REST/streamers/get-all-contexts/

Action: Returns the list of all available story contexts – used to choose which story to watch on the story server.

• *scheduleContinuousStory()*

http://[story-server:port]/REST/streamers/schedule-continuous-story/{ip}/{selected}/{timetostart}

Parameters:

- *maxHarmonization* - The maximum number of choices to try to harmonize each interaction cycle

- *votingStrategy* - The name of the chosen voting strategy

Action: Schedules the selected context to be dramatized on the selected Drama Streamer interface. Since this is a multiuser system, the act of scheduling and joining a story to watch it are not necessarily single user exclusive.

• *getTimeToStart()*

http://[story-server:port]/REST/streamers/get-time-to-start/{storyid}

Action: Returns the time in milliseconds that a given story will still take to start, according to its schedule.

• *getCurrentChapter()*

http://[story-server:port]/REST/streamers/get-current-chapter/{ip}

Action: Returns the current chapter being shown on the Drama Streamer. This is important since the story client may need to get different data from the chapter and its events other than just the story visualization, which is provided by the video stream.

• *getSuggestions()*

http://[story-server:port]/REST/streamers/get-suggestions/{ip}

Action: Returns the available suggestions for the story (and chapter) being watched on the Drama Streamer. This is basically the main interaction mechanism in this version of story, where a user can vote for what he or she wants to happen in the story – this list is built by the system based on its logic [5]

• *requestSuggestionInsert()*

http://[story-server:port]/REST/streamers/request-suggestion-insert/{ip}/{suggestionid}

Optional parameters:

- *userlogin* Registered user encrypted login

- *userpassword* Registered user encrypted password

Action: Inserts a vote for the selected suggestion for the story being watched on the selected Drama Streamer. Since this is a multiuser system, more than one user can vote for different suggestions, where the most voted will be chosen and

then be used for the following chapters of the story being watched. Note that the user is optional, the main advantage of a registered user is being able to keep a history of votes, which may lead to stronger votes, depending on the voting strategy.

4.5. Evaluation and Tests

In order to assess the effectiveness of the presented work, tests were done using a partial model implementation. For this objective, different aspects of the model were evaluated, like the performance and architecture, and the voting strategies.

4.5.1. Performance and architecture

To evaluate the performance requirements of the proposed model, tests were to done to measure the average delay time observed according to the number of users connected. These tests were done using a single node (a Drama Server capable of rendering and streaming the story) in order to test the capacity of the proposed multiuser system to handle loads without too much performance degradation. Since the model is based on a scalable architecture (JBoss), it can be argued that by doing these tests on a single node, and reaching satisfactory results, that the model's implementation show that all the requirements are met, as in this scalable architecture more nodes can be added seamlessly to create a cluster and attend to more users.

In the prototype, we used an i5 notebook running Windows 7 with 6 GB of RAM as the basic server machine. We implemented two separated servers running on the same machine, one working as a Story Server and the other one as the Drama Server. As clients, we used the following machines: (1) a 7-inch Android tablet running Ice Cream Sandwich with 1 GHz CPU and 1 GB RAM; (2) a Windows XP 1 GHz Netbook; (3) a Smartphone with 1 GHz CPU and 384 MB RAM.

With 1 Drama Server	1 User	2 Users	3 Users
Delay	3.2 seconds	3.3 seconds	3.4 seconds

Table 5 Performance with multiple clients on VLC Streamer

Results on Table 5 demonstrate the short delay present when streaming stories in the prototype implementation. Most of this delay is caused by the CPU intensive process of rendering the 3D story on the Drama Server, together with the live transcoding of the stream. On a real production system, better computers can be used. Moreover, even with a 3 second delay, it should be noted that this test reveals one of the most positive aspects of the proposed model: the same quality of story rendering would be much harder to achieve if all the story client implementations had to render the story by themselves.

Tests on the Twitch platform using the Twitch Streamer implementation had a higher average delay of 40 seconds, that could also be reduced by using better hardware. However, since Twitch already supports thousands of simultaneous users, as mentioned in this chapter, this delay may be acceptable, as there is a lesser need for the amount of Drama Server clusters. Moreover, Logtell stories can have a long enough duration in order to have these delays as acceptable: interaction is not supposed be in real-time.

Further benchmarks were done to evaluate the performance of the REST interface of a single cluster when receiving multiple requests to the server API. These tests were done using simulated multiple concurrent requests, calling the same methods that return, for instance, the list of available streamears, story suggestions and chapter descriptions, needed for the clients interface:

- Time taken for tests: 22.956 seconds
- Complete requests: 10000
- Failed requests: 0
- Requests per second: 435.61 [#/sec] (mean)
- Time per request: 2.296 [ms] (mean, across all concurrent requests)

These results show that once again, a single node is very capable of receiving an adequate number of clients. As before, these nodes can be multiplied, thus scaling according to needs.

4.5.2. Voting Strategies Tests

In this section, we can see how some of the possible voting strategies work. These tests demonstrate how the proposed voting strategies behave according to users votes while watching stories. These tests were done by actually using the prototype implementation and testing the same stories with different voting strategies. As shown, they reach their intended goals: to consider votes differently, according to the desired level of power that each vote can represent.

As noted, some of the voting strategies can be too unbalanced or unfair (if so deemed by the controlling users), and for that the balanced strategies comes to help by trying to reach a reasonable equilibrium of effective voting power. By doing so, these strategies can, for example, avoid Dictators, that can always decide no mater what the whole group of voters want together.

The tests of the voting strategies show how the different suggestions, which are at first gathered in a random way, are returned in a sorted/ranked list, to be used in the story suggestion insertion process. This, together with the harmonization of suggestions, can be used to create stories that consider the users interests and adapt to them.

For simplicity, assume that 'sug *x*', where *x* is a number, is a suggestion inserted by a user in a given story during a chapter's voting phase (ex: "Draco kills Marian", "Brian fights Hoel", etc). Also, assume that whenever a suggestion has a user in parenthesis, it means that this is a vote from that user. Ex: "*sug 1 (user 1)*" means that it is a vote for suggestion "sug 1" from user "user 1". When there is no user assume it is a anonymous / new user, without a history of votes, so the user is omitted.

SimpleUserChosenVoteScoreVotingStrategy sample voting:

User 1 chosen vote score = 2.0

User 2 chosen vote score =3.0

Total suggestions = [sug 1, sug 1 (user 1), sug 2, sug 3, sug 3 (user 2)]

The most voted suggestion has 4.0 score.

Ranking: [sug 3, sug 1, sug 2]

SimpleUserGoodVoteScoreVotingStrategy sample votation

User 1 good vote score=3.0

User 2 good vote score=2.0

Total suggestions = [sug 1, sug 1 (user 1), sug 2, sug 3, sug 3 (user 2)]

The most voted suggestion has 4.0 score.

Ranking: [sug 1, sug 3, sug 2]

WeightedChosenGoodVotingStrategy

User 1 chosen vote score =3, good vote score= 1 total user vote weight=5.5

User 2 chosen vote score =3, good vote score= 2 weight=6.5

Total suggestions = [sug 1, sug 1 (user 1), sug 2, sug 3, sug 3 (user 2)]

The most voted suggestion has 7.5 score

Ranking: [sug 3, sug 1, sug 2]

When working with the balance square root strategy, before votes are even counted, the users have their voting power balanced by our square root, inspired by Penrose's work [40]. In the rest of this section, there are some scenarios that show that we reach the desired situation, by not allowing voters to have full control of the situation, while keeping them more powerful than new voters.

ProbabilityVoteStrategy

By running some tests, we can see that this strategy promotes a random result while giving more chances to popular choices.

Suppose a set of suggestions where each one has equal votes(2):

Total suggestions = [sug 1, sug 1, sug 2, sug 2, sug 3, sug 3]

By running 1,000,000 interactions, we can see that results follow the expected distribution of probability (1/3). In this ex:

Victories=

Sug 1 - 333801

Sug 2 - 333087

Sug 3 -333112

Even if we use more interactions, they also keep regular. With 10,000,000 interactions:

Victories:

Sug 1 - 3332076

Sug 2 - 3333189

Sug 3 - 3334735

This shows that the implementation is "random" enough, since these results show that they follow the expected distribution even with big samples.

However, if we use a set of votes in which one of the suggestions has more votes, we see that they still have more chance of being picked. For instance, using 1,000,000 interactions with the set (where 'Sug 2' has 3 votes while the others have 2 votes each):

Total suggestions = [sug 1, sug 1, sug 2, sug 2, sug 2, sug 3, sug 3]

We get the results

Victories:

Sug 1 - 285871

Sug 2 - 428510

Sug 3 - 285619

Thus showing that the voting method works as intended, since 'Sug 2' wins more often with a large number of interactions.

BalancedSqrtWeightedChosenGoodVoting

In this voting strategy, multiple steps of weight adjustment must be done. Here some results show how the model handles situations in which the voters have different voting power.

Table 6 presents a list of the partial steps of the vote balancing, showing how the algorithm balances votes for each iteration until they are balanced enough.

Situation 1:

State	Description
Initial	user 1 chosen vote score =30, good vote score= 1 weight=46.0 user 2 chosen vote score =3, good vote score= 1 weight=5.5 total sugs=[sug 1, sug 1 (user 1), sug 2, sug 2, sug 2, sug 3, sug 3 (user 2)] top weight = 46.0 sumWithoutTop/2= 4.25 (not balanced)
Balancing	new set: [SuggestionAndWeight [suggestion=sug 1, weight=1.0], SuggestionAndWeight [suggestion=sug 1 (user 1), weight=6.782329983125268], SuggestionAndWeight [suggestion=sug 2, weight=1.0], SuggestionAndWeight [suggestion=sug 2, weight=1.0], SuggestionAndWeight [suggestion=sug 2, weight=1.0], SuggestionAndWeight [suggestion=sug 3, weight=1.0], SuggestionAndWeight [suggestion=sug 3 (user 2), weight=2.345207879911715]] top weight = 6.782329983125268 sumWithoutTop/2= 2.6726039399558577 (not balanced yet)
Balancing	new set: [SuggestionAndWeight [suggestion=sug 1, weight=1.0], SuggestionAndWeight [suggestion=sug 1 (user 1), weight=2.604290687140218], SuggestionAndWeight [suggestion=sug 2, weight=1.0], SuggestionAndWeight [suggestion=sug 2, weight=1.0], SuggestionAndWeight [suggestion=sug 2, weight=1.0], SuggestionAndWeight [suggestion=sug 3, weight=1.0], SuggestionAndWeight [suggestion=sug 3 (user 2), weight=1.531407156804393]] top weight = 2.604290687140218 sumWithoutTop/2= 2.2657035784021966 (not balanced yet)
Balancing	new set: [SuggestionAndWeight [suggestion=sug 1, weight=1.0], SuggestionAndWeight [suggestion=sug 1 (user 1), weight=1.6137814868005576], SuggestionAndWeight [suggestion=sug 2, weight=1.0], SuggestionAndWeight [suggestion=sug 2, weight=1.0], SuggestionAndWeight [suggestion=sug 2, weight=1.0], SuggestionAndWeight [suggestion=sug 3, weight=1.0], SuggestionAndWeight [suggestion=sug 3 (user 2), weight=1.237500366385559]] top weight = 1.6137814868005576 sumWithoutTop/2= 2.1187501831927795

Table 6 Balancing Weights

Voters weights are balanced. The 1 most voted suggestion have 3.0 score:

Ranking =[sug 2, sug 1, sug 3]

Situation 2:

Now suppose there would be a tie, with the same users but only 2 votes for suggestion 'sug 2'. In this case the balanced weight will be a good tie breaker.

State	Description
Initial	total sugs=[sug 1, sug 1 (user 1), sug 2, sug 2, sug 3, sug 3 (user 2)] top weight = 46.0 sumWithoutTop/2= 3.75 (not balanced yet)
Balancing	new set: [SuggestionAndWeight [suggestion=sug 1, weight=1.0], SuggestionAndWeight [suggestion=sug 1 (user 1), weight=6.782329983125268], SuggestionAndWeight [suggestion=sug 2, weight=1.0], SuggestionAndWeight [suggestion=sug 2, weight=1.0], SuggestionAndWeight [suggestion=sug 3, weight=1.0], SuggestionAndWeight [suggestion=sug 3 (user 2), weight=2.345207879911715]] top weight = 6.782329983125268 sumWithoutTop/2= 2.1726039399558577 (not balanced yet)
Balancing	new set: [SuggestionAndWeight [suggestion=sug 1, weight=1.0], SuggestionAndWeight [suggestion=sug 1 (user 1), weight=2.604290687140218], SuggestionAndWeight [suggestion=sug 2, weight=1.0], SuggestionAndWeight [suggestion=sug 2, weight=1.0], SuggestionAndWeight [suggestion=sug 3, weight=1.0], SuggestionAndWeight [suggestion=sug 3 (user 2), weight=1.531407156804393]] top weight = 2.604290687140218 sumWithoutTop/2= 1.7657035784021966 (not balanced yet)
Balancing	new set: [SuggestionAndWeight [suggestion=sug 1, weight=1.0], SuggestionAndWeight [suggestion=sug 1 (user 1), weight=1.6137814868005576], SuggestionAndWeight [suggestion=sug 2, weight=1.0], SuggestionAndWeight [suggestion=sug 2, weight=1.0], SuggestionAndWeight [suggestion=sug 3, weight=1.0], SuggestionAndWeight [suggestion=sug 3 (user 2), weight=1.237500366385559]] top weight = 1.6137814868005576 sumWithoutTop/2= 1.6187501831927795

Table 7 Balancing Weights 2

Now voters' voting power is balanced enough. Balanced user weights:{1=1.6137814868005576, 2=1.237500366385559}

The 1 most voted suggestion has 2.6137814868005576 score (summing all votes considering the new user weights), so there is no tie: Ranking =[sug 1, sug 3, sug 2]

4.6. Conclusions

Analyzing the test results, it seems fair to conclude that the model shows promising results towards the desired behavior of the multiuser system, where the Drama Server provides streams to multiple story clients and it is in charge of

heavier CPU loads than those found in the clients. Also, the model's voting strategies allow voters to have the desired amount of power: whether that is rewarding experienced voters with "full" history voting power, or balanced voting weight.