SYNCHRONIZATION ALGORITHM FOR PEER-TO-PEER INTERACTIVE DISTRIBUTED SIMULATION IMPLEMENTED IN A SINGLE-THREADED WEB APPLICATION

Štěpán Karták

University of Pardubice, Faculty of Electrical Engineering and Informatics

stepan.kartak@student.upce.cz

ABSTRACT

This article presents a synchronization algorithm for distributed interactive peer-to-peer simulation in a web browser. It is a practical utilization of web browsers in combination with modern technologies for realization of computation- and network-heavy simulation tasks. Due to present limitations of web browsers, especially due to their limited computing power, a class of realizable tasks is defined. Such tasks can be successfully solved with web browsers. The article also covers the operating principles of web applications, with focus on JavaScript and problems arising from its concept. The text describes the algorithm and used network topologies of the logical processes, and their synchronization methods. Advantages and disadvantages of realizing a simulation with a web browser are described as well, as well as the reasons why the not-so-frequently used peer-to-peer simulation was used. To conclude, a use case for application scope testing is characterized, i.e. identifying the appropriate number of logical processes, frequency of interactive interventions, usable numbers of objects, etc., for which the presented algorithm is appropriate.

Keywords: Distributed Simulation, Web-based simulation, HTML5, WebRTC

1. INTRODUCTION

This article is focused on using a web browser to realize a user-friendly interactive distributed simulation. The goal is to design and create a relatively general algorithm (in terms of defined class and application scope), which would provide the user with basic functionality (simulation core, synchronization, interactive approach, …) to realize simple simulations. The user does not have to solve the basic implementation problems in the limiting environment of a web browser, and can focus on the implementation of the simulator behavior.

Today, web browsers are very well suited for such applications. Since the year 2012 (Karták 2014), web browsers have offered functionalities that allow realization without the use of third-party plugins. However, even in spite of the significant advances made by web browsers in the last few years, the realization of a distributed simulation is not possible without a number of compromises.

The core of the solution is based on the previous work on distributed web simulations (Kartak 2015, 2016), which focused on trainer applications, that is for applications for testing (examination, education) of workers / dispatchers and distributed web simulations in general. This solution served as a proof of concept, and the article expands the concept.

2. WEB APPLICATIONS AND TECHNOLOGIES

A web application from our point of view is considered to be distributed from the web server (in a certain configuration) through a computer network (local or global Internet network) into the client device, which is an instance of a web browser window. The application is distributed statically (an HTML description of the page structure is downloaded from the server, as well as additional information – uncompiled JavaScript code, CSS files, images, fonts etc.). After loading all the static parts listed above, the dynamic part of the web is launched – the program code written in JavaScript. This code is distributed to the client computer uncompiled, and is run according to the requirements of the author of the code.

The program of the webpage reacts to user interaction (typically mouse and keyboard) or external information (application state update by the server, information and events provided by the web browser itself).

All the actions stated above are called events (js event), and are executed sequentially, in the chronological order of their creation. The order or priority of their processing cannot be influenced on the browser level. This approach is suited for primarily static web pages with minimal amounts of program code or time-consuming calculations. When realizing heavily interactive applications, such as interactive distributed simulations, this approach is generally inappropriate (see chapter 4.4).

Web applications are always downloaded from a web server, which may or may not participate on further

client-side runtime. The server is often only a source of static content, and does not store any user states, or provides only elementary functionality, such as user identification, and is not informed of any further states of the client-side, or the server receives only the results of the algorithm that was run on the client-side, with no interaction or interruptions by the server after the static content is downloaded. Sending the results of user or script activity is typical for web applications.
Web applications can transfer workload to the client-side (web browser). From this point of view, a browser may be considered a thick client as well as thin client, depending on the used approach.

3. CHARACTERISTICS OF APPLICATIONS IN THE TARGET DOMAIN OF THE ALGORITHM

In this chapter, used terminology is stated, along with a basic introduction to the algorithm.

First, it must be stated that the algorithm assumes use of discrete distributed simulation, that is to say that the behavior of applications for which the algorithm is suitable must be determined by discrete events.

3.1. Basic terminology

As stated above, the simulation runs in web browsers. One specific instance of a web page (in a web browser) represents one logical process (LP). A group of logical processes forms a distributed simulation.

The algorithm primarily works with three elements:

- **Entity** ... in fact an object passing through the simulation,
- **Activity** ... an event handling procedure,
- **Simulation event** ... a planned discrete event of a concrete activity.

Interactive interventions are realized as interruptions of discrete planned activities.

3.2. Application aspect

Use of web browsers presents an inexpensive way of realizing distributed interactive simulation. A typical user might be a small company, requiring a trainer simulator or training software, that can be described by an algorithm of discrete events.

The introduced algorithm is designed for three application classes:

1. **Trainer simulator applications.** Assume a group of workers that forms a single team solving a problem or reacting to a chosen situation. Every worker/user works with a browser, where he or she observes a simulation scene, and interacts with the simulation runtime within the frame of the assigned logical process.
   Another example may be a railway station dispatcher in the scope of a region (where there are more dispatchers). Another example may be the simulation of a technological process / production, where every employee is responsible for a part of the process.
2. Realization of a **simple multiplayer game**, where logical processes represent the space for individual players, with implicitly shared state-space of the simulation (the game environment). This application class is covered by the use cases (chapter 7).
3. **Distributed space for data exchange** within a work group. This application class does not directly represent a simulation. Only the synchronization methods are used to keep the memory space up to date for all of the logical processes.

The primary application class can be generally classified as a distributed system requiring interactive approach, based on discrete events and with no complex calculations present.

3.3. Networking aspect and topology of the logical processes

This solution utilizes primarily web browsers, that contain a majority of the simulation calculations. The server part is not present in the calculations or logic, and serves only for undemanding secondary activities (initialization of the connection between clients, creation process of the simulation, etc.).

The solution is a purely peer-to-peer simulation.

This solution was chosen due to the fact that web browsers are commonly found on computers, and nothing prevents their participation in simulations. The opposing server architecture (for example the commonly used HLA architecture with federates running on servers) requires high-end (and expensive, or not widely accessible) servers (Kuhl et al. 2007).

Creating a peer-to-peer network of clients allows to tap into the potential of the client computers and, at the same time, requires no extra expenses such as powerful servers or software. With web browsers, there are usually no connectivity issues in terms of firewall limitations and similar problems, as web browsers activities are generally considered safe, due to the sandbox nature of the browsers themselves.

This solution however has certain disadvantages as well. The most significant disadvantage is the considerably lower performance in comparison to desktop applications. This is another reason why direct connection between the clients is beneficial, as opposed to communicating through a server. The sent message travels directly to the target client, instead of two messages being sent (client-server and server-client). If we consider a local network (low latency, high bandwidth), the theoretical time needed to deliver a message from client to client when using a server

architecture is double the time needed in a peer-to-peer architecture.

3.4. Implementation scope of the synchronization algorithm

The introduced algorithm is a general algorithm that solves the synchronization problems of logical processes.

The implementation of the algorithm solves the following (details in chapters 6 and 7):

- discrete simulation core (primarily the event calendar queue),
- events,
- prototypes of logical processes,
- prototypes of discrete activities and auxiliary discrete activities (running in the background for algorithm needs),
- prototypes of entities,
- synchronization of logical processes,
- handling of interactive user input,
- elementary handling of entity collisions,
- (optional) rendering of a simple 2D scene,
- (optional) simulation interruption handling when waiting for user input.

The following is not solved by the algorithm:

- specific implementation of the target application,
- specific logical processes,
- specific simulation activities,
- specific entities,
- specific collisions (of entities) and exceptions of interactive user input handling.

The primary goal is to provide an algorithm that implements the necessary structures and solves the above stated problems for the user, allowing him or her to concentrate on the process of modelling the solved use case – the logic (activities) and objects (entities), and their interactions.

The aim of this work is not to create competition for extensive standards such as DIS, HLA, TENA etc. (IEEE 1278.1-2012; Kuhl et al. 2007), and similar, as that is, due to the limitations (see chapter 4.4) of web browsers, impossible.

3.5. Reusability of the solution

The aforementioned algorithm will be available as a JavaScript function library, which shall implement the following functionalities:

- Connection of a logical process into the administration interface,
- synchronization of a running simulation,
- basic functional support for realization of animated output.

4. USED TECHNOLOGIES

Web distributed simulation could not be realized without new technologies, collectively referred to as HTML5. These functions expand the capabilities of web browsers with functions that were formerly the domain of desktop or server applications (a typical example would be two-way network communication), and achieving the desired effect before HTML5 required use of third-party plugins (typically Java applets), or inefficient solutions (an example might client's periodical queries about state changes, instead of direct "state changed" notice sent directly from the server to the client).

An enumeration of the fundamental HTML5 technologies, on which this solution is based, of follows.

4.1. WebRTC

The WebRTC technology servers to connect clients (instances of browser windows) directly, without the need to use a server as a connecting link. This technology is primarily used for peer-to-peer sound and video transmission (typically videoconferences). However, pure data transmission is implemented as well, which can be used to send user data, and is fundamental for the algorithm's operation.

4.2. WebSocket

This network technology serves to create a permanent (until the browser window is closed) two-way server-client connection. The client may be informed of the server state changes directly by a message from the server, bypassing the need to periodically ask the server "Are there any news?". The second, equally important benefit is the persistent client-server connection. When sending messages, it is no longer necessary to create the connection every time. This can save up to tens of milliseconds, depending on how busy the server is.

4.3. Canvas

The HTML tag *<canvas />* defines area for 2D drawing. Thanks to this HTML element, a drawing area of any size can be created, and drawn upon using JavaScript.

However, the *<canvas />* tag does not allow the scene to be partially redrawn. This is a limiting factor. Depending on the size of the drawing area and the number of the rendered objects (generally graphic primitive types) the time needed to redraw the scene may increase significantly.

The WebRTC, WebSocket and Canvas technologies are the fundamental building stones of the web simulation realization.

4.4. Basic characteristics of JavaScript

The JavaScript programming language, generally used by web browsers to realize dynamic behavior of web

pages, is a weakly-typed prototype language. JavaScript also contains several functions which significantly complicate optimization of compiled code, which in turn means that the resulting program performs significantly worse than desktop applications. However, this state improves with new versions of web browsers. Optimization libraries (such as the asm.js) exist, which compile C/C++ language code into strongly optimized JavaScript code (Voracek 2016). However, this approach does not solve the second, more significant, problem – the fact, that JavaScript has a "single-threaded" (this expression is not completely accurate, but captures the essence of the problem, which is why the expression will be used further in the text) approach.

A single-threaded event-based system of executing user code is a critical problem for a distributed simulation-type application. This property of JavaScript means that in practice, all operations are executed synchronously in a single thread. There are no concurrent multi-threading approaches available to the user (Processes that cannot be influenced by the user, such as data rendering, network communication etc. are, however, done asynchronously by the browser). This state does not present a problem to a large group of algorithms that are commonly realized in a browser, but generally complicates construction of algorithms for tasks that require parallel execution of operations (in terms of multi-threading and multi-processor execution), be it for effectiveness or individual tasks' time complexity reasons. Chapter 6.1 covers the specific reasons because of which this state presents a significant problem to realization of web (distributed) simulations.

5. AUXILIARY SOFTWARE SOLUTIONS

As stated before, the presented solution assumes the simulation runs only in a web browser (with no participation from the server), purely peer-to-peer.

However, no peer-to-peer solution is capable of bypassing the server side completely. At the very least, connection initialization must be solved, which is impossible without the participation of a sever element. If we want to conduct distributed simulation, we have to build it somewhere, or at least save the configuration (for example on a web server), from where it will be available to the clients. Due to the fact that the presented solution works with up to 40 connected clients, it is necessary to observe the state and behavior of the client computers.

There are 4 auxiliary server applications, connected into the Administration interface, which runs in the "background" of the simulation itself:

5.1. Model configuration

The fundamental part of the server-side Administration interface of this solution. Serves to register individual types of logical processes and consequentially use then when building the model of the distributed simulation.

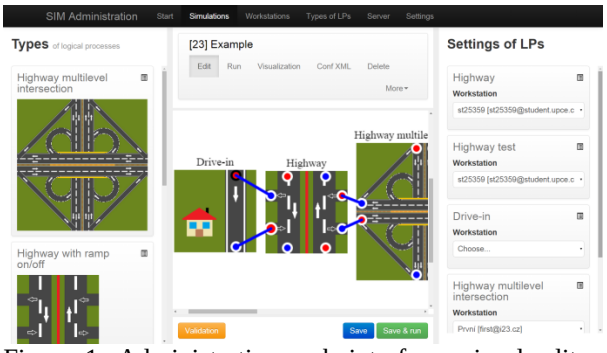

Figure 1: Administration web interface, visual editor; blue lines are network connections between logical processes (chapter 3.2 part 1)

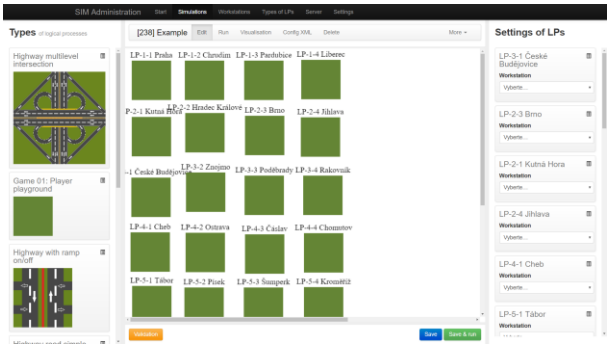

Figure 2: Administration web interface, visual editor; 20 logical processes represents 20 players (chapter 3.2 part 2 and use case – chapter 7)

5.2. Simulation control

A server-side application to which all the clients – logical processes – are connected. Through this application, it is possible to pass commands and instructions or gather runtime information from clients (in bulk). This application serves to control the simulation (initialization, start, pause, end, etc.) in a centralized fashion.

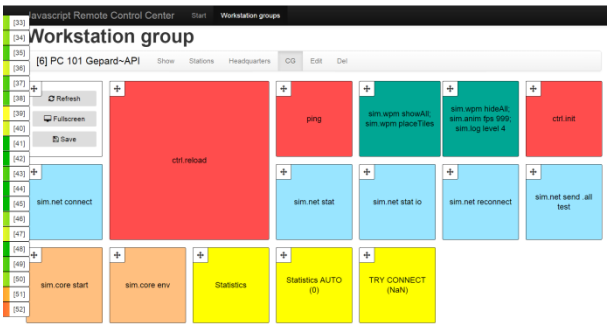

Figure 3: JSRC: Prepared command set, one square is user-defined command (or commands), prepared for touch-devices

5.3. Centralized visualization

This server part (realized as a component of the Administration interface) facilitates recording of the animation output. Screenshots for static preview (see figure 4) of the logical processes' state are captured, as

well as the animation activities, making it possible to reconstruct the logical processes and simulation runtime.

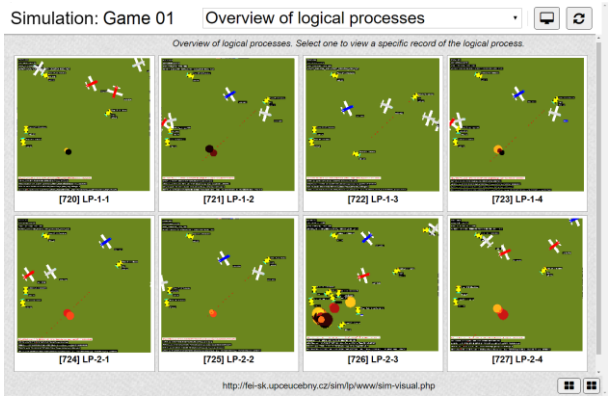

Figure 4: Overview of logical process in simulation in central visualisation, 8 LP/player, shared space/playground, differences between pictures are caused by the creation of a screenshot at different times

5.4. Initialization server

The initialization server serves to create a peer-to-peer connection between browsers. This component also graphically depicts the state of connections between the individual clients.

6. ALGORITHM CHARACTERISTICS

6.1. Simulator as a single-threaded application

A simulator faces four critical tasks, which are independent and running at all times: (i) simulation core, executing discrete events, (ii) network communication (sending and receiving messages) with other simulation participants, (iii) reactions to user input and (iv) animation output.

These four parts are commonly realized as parallel tasks in classic desktop applications. In JavaScript, this is not possible. This is why the simulator is realized as a series of cyclically repeated operations (only fundamental steps are listed):

1. Interpretation of incoming messages.
2. Interpretation of user input.
3. Synchronization of the logical process, based on steps 1 and 2 (see chapter 6.3).
4. Execution of available (especially in relation to synchronization of logical processes) discrete events.
5. Calculation of animation output:
   (a) entity position calculation,
   (b) calculation of collisions or other interactions between entities,
6. Broadcast data (local LP state information) to other logical processes.
7. Rendering of the situation onto the animation output.
8. Continue by step 1.

6.2. Basic structure of logical processes

A logical process is made up of 6 parts (for an UML diagram see Image 5):

1. Simulation core: operates simulation activities, ensures synchronisation. Includes:
   (a) Calendar: priority queue for simulation activity planning.
   (b) Environment: contains environment and state information related to the simulation (primarily activity handler).
   (c) Modules: any named data structure, usually auxiliary, available to all dependent parts (usually activity handler). Used, among others, for the text report of simulation states.
2. Simulation activity: specified the type of activity, time of execution and any other additional information
3. Activity Handler: execution of given activity type
4. ConnectionRegister: logical process communication realisation layer
5. Animation Activity: Described a graphic element for animation rendering. One of the modules of the simulation core.
6. AnimationManager: renders a scene based on the animation activities

Other program parts that are not critical for the execution of a logical process:

7. SettingsManager: contains a description of the simulation configuration.
8. EntityManager: contains information about entity types and individual entities.
9. ActionManager: describes interactions and eventual reactions of individual entity types.

The solution as a whole works under several basic premises:

- All simulation and animation activities can be serialized.
- All simulation and animation activities can be interrupted at any time (removed from the queue or scene).

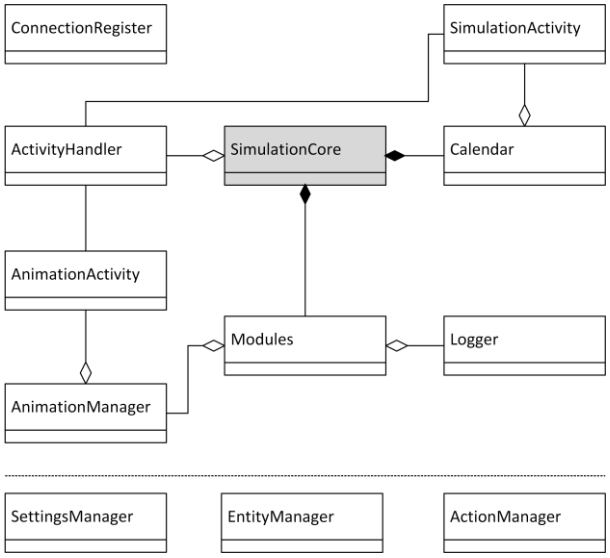

Figure 5: Basic UML schema of logical process (the simulator itself)

The algorithm stated above is only a simplified framework of the used solution, Provided for a basic understanding of the function.
Moreover, the problem presented by the event-based approach of JavaScript is not solved. That means that at any given time, the execution of the algorithm may be interrupted by executing different tasks (user input into the simulation, receiving messages, etc.). The event-based approach is not a problem in general, as the "main" algorithm will continue after the interrupting operation is finished. The problem is that these "unexpected" actions take some time to complete, and thus slow down the calculations and have a negative impact on the animation smoothness, which in the end means worse user experience.

6.3. Topology of connection between logical processes in the simulation model

A one-on-one connection is realized (through the WebRTC technology). During the initialization process of the simulation, a connection is made between each logical process. The connection is established primarily in order to maintain a global memory space. All state changes of a logical process are sent to all other logical processes (in fact a broadcast of state changes). Every receiving logical process then decides whether and how to process the received data. This process was inspired by the DIS standard.
This broadcast of changes between all logical processes is also used for synchronization purposes.

6.4. Logical process synchronization

Optimistic methods of synchronization are generally more suited for interactive simulation, as they do not require strict time synchronization of the logical processes runtime, which in turn means the calculations (and animations as well) are smoother (thanks to not having to wait for the "slow" logical processes). To ensure smooth operation (especially in terms of animation), the conservative approach is not effective, as it requires a short look-ahead (briefly: max. look-ahead must equals to delay of animation slides – animation FPS 25 required 40 ms between slides / 40 ms look-ahead) to ensure smooth animation, which increases communication load.

A "two-level" synchronization method was chosen:

1. For basic synchronization, the Conservative synchronization technique of sending null messages with a look-ahead (Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm, Fujimoto 2000) was used – the specific implementation can be found in the previous work (Kartak 2015). This method is utilized primarily in during the simulation initialization, and to capture above-average fluctuations (delays) in network communications.
2. For precise synchronization purposes, state information timestamp readings are used, as sent by other logical processes. The received times are compared to the actual system time of the client computer, and based on the differences of the other logical processes and the client logical process (and its anticipated behavior), the speed of the logical process is adjusted – and with it, the animation speed, as it is animation speed that determines the speed of the simulation.
   In this second level of synchronization, the strict time synchronization with conservation of local causality of time is *not* applied. We assume a deviation (depending on the scope of the simulation model) of up to 100 ms. We consider the deviations in this interval to be negligible, and (nearly) imperceptible by the user.

Due to the facts stated above, use on local (e.g. company) networks is presumed, where the latency of messages sent through the WebRTC is usually around 10 ms, which allows smooth runtime of the application.

6.5. Algorithm

We are in a web browser / JavaScript environment, a single thread event based approach.

Initialization

1. Create (WebRTC) connection between all LPs. *n* logical processes make
   $$n \times n - n$$
   full-duplex connections
2. Waiting until a every connection is established

Start of simulation

Every one logical process

1. Sync level 1: Conservative synchronization technique (see chapter 6.4)

ISBN 978-88-97999-85-0; Affenzeller, Bruzzone, Jiménez, Longo and Piera Eds.

2. Waiting until all logic processes are obstructed to run and initialization is not performed. (For example: waiting 5000 ms)

Running simulation (one logical process execution), inc. sync level 2

Every one logical process
SC ... Simulation core
A ... Animation
A.ActivityList ... list of animation activities
E ... simulation event
SC.E ... current (executing) event
SC[C] ... calendar of events/activities

DECLARE SC.run
BEGIN
1. SC.executeUserAndNetworkEvents()
2. isAnimTimeInFuture = A.time>=SC.time
3. IF isAnimTimeInFuture
   THEN
   1. WHILE SC.time<=A.time
      THEN
      1. IF SC[C].isEmpty()
         THEN A.start(); follow step 1
         ELSE
         1. follow step 9 to 11

2. IF SC[C].isEmpty()
   THEN A.start(); follow step 1
4.
5. IF SC.time>A.time
6. THEN
   1. A.setTime(SC.time)
   2. A.setSpeedRatio(1)
   ELSE
   1. A.setSpeedRatio(0.98)
7. A.start()
8.
9. SC.E = Shift first E from CS[C]
10. SC.time = SC.E.time
11. e.execute()
12. IF SC[C].nextEvent().time == SC.time
    THEN follow step 3
    ELSE follow step 8

DECLARE A.start
BEGIN
SC.executeUserAndNetworkEvents()
stepStart = NOW.time
IF A.stepLastTime != 0
THEN timePlus = stepStart − A.stepLastTime
ELSE timePlus = 200 # magic constant for first step
timeAdd = timePlus*A.speedRatio*0.98 # 0.98 is a constant defining a delay in the execution of the script itself, experimental value
A.time += _timeAdd
A.stepLastTime = stepStart
WHILE A.ActivityList.hasNext()
1. AACurrent = A.ActivityList.next()
2. AACurrent.timePrepare(A.time) # Calc new position
SC.collisionCalculation()
WHILE A.ActivityList.hasNext()
1. IF AACurrent.getStartTime()>A.time
   THEN continue;
2. AACurrent.draw(A.time) # (re)draw activity to output buffer
A.outputFrame() # Render to screen
WHILE A.ActivityList.hasNext()
1. AACurrent = A.ActivityList.next()
2. IF AACurrent.isFinished(A.time)
3. THEN A.ActivityList.remove(AACurrent)

animNextDiff = 1000 / A.fps # requested FPS
timeAnimDuration = NOW.time − stepStart
timePlanPlus = animNextDiff − timeAnimDuration
IF timeNextAnimX<0
THEN timePlanPlus = 2
plan(A.start, timePlanPlus)
# plan A.start() by x ms, where timePlanPlus is demanded delay between now and next output frame (by requested FPS)
END

A.time = SC[C]. nextEvent().time # Setup time of animation output, example
SC.run()

SC.broadcastStateInfoInterval(ms=30) # Send state info to all another LPs every 30 ms, this is realized as standard E planned every *x* ms

SC.initMessageReceiver(# Receive message event
(scSender.name, listOfStateInfo) =>
stateInfoEv = new E
stateInfoEv.listOfUpdates = listOfStateInfo
SC[C].addEvent(stateInfoEv, SC.time+1ms)
)

7. USE CASE AND TRACKED METRICS

Algorithm and implementation of distributed simulation in a web browser were tested on a game type program (see figures 6 and 7):

- Each LP contains a single user-controlled entity (UCE).
- One shared scene, representing the playing field (all logical processes share a single scene i.e. all users see the same).
- The playing field will be restricted by screen size (the area of the canvas is 1 MPx),
- The user controls the UCE with a keyboard (arrows allow movement in 4 basic directions, spacebar allows the user to shoot) and a mouse (click into the playing field represents a travel destination), figure 8.
- A shot (realized as an entity) travels with a limited speed, giving the remaining users time to react (figure 9).

- The collision of a shot with a soldier causes an action (frag count, unimportant for the use case), figure 10.

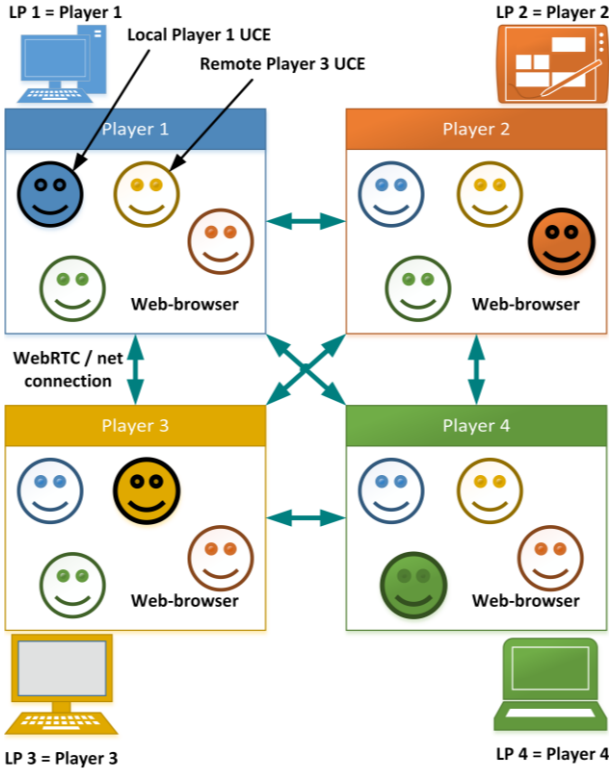

Figure 6: Conceptual picture of use case, LP topology

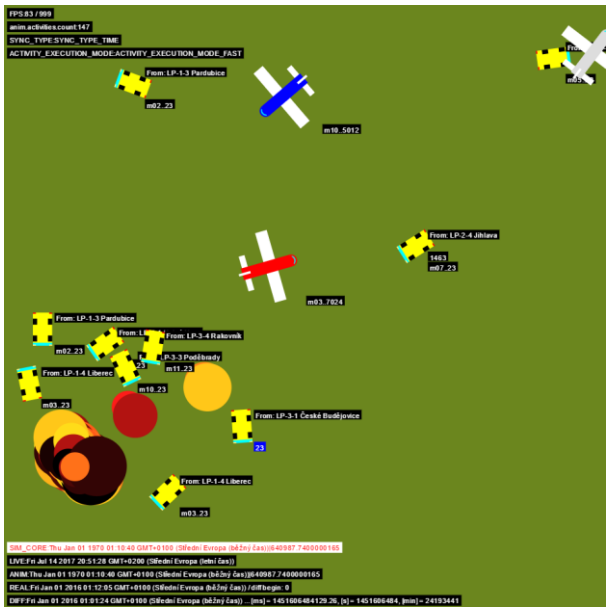

Figure 7: The screenshot from use case – a simple multiplayer game

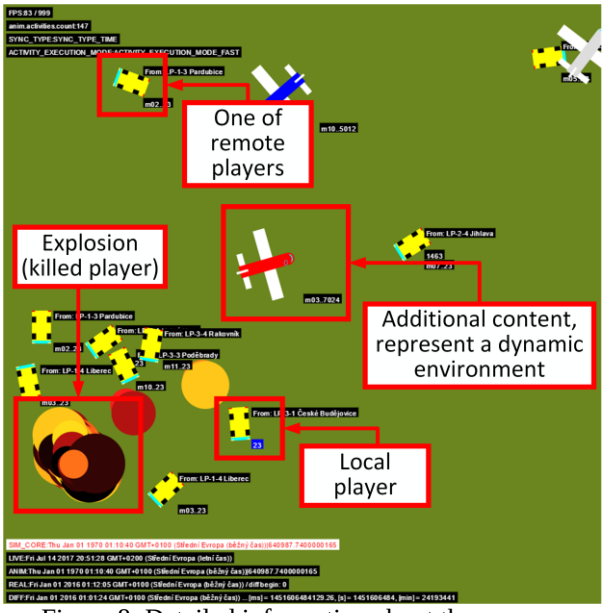

Figure 8: Detailed information about the use case

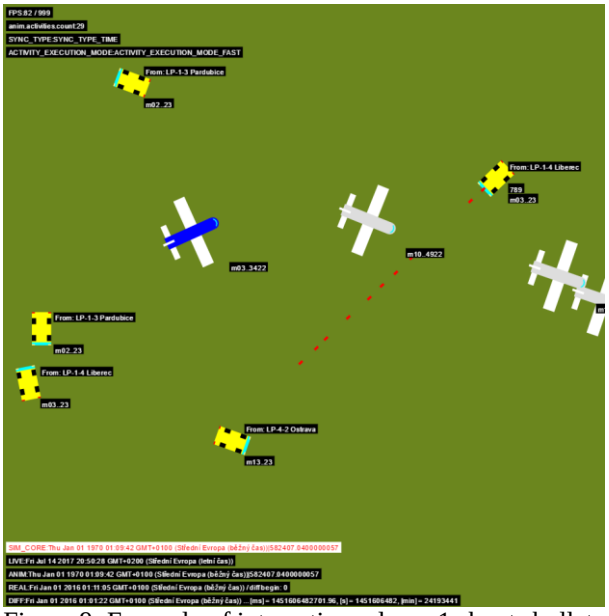

Figure 9: Example of interaction, player 1 shoots bullets

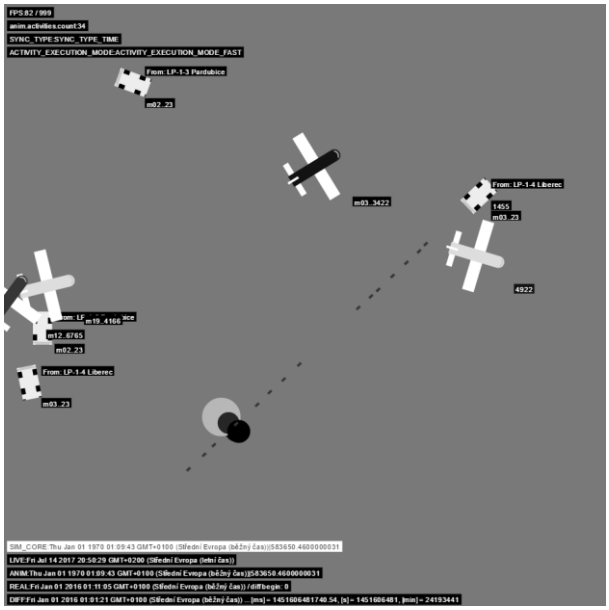

Figure 10: Example of interaction, player 2 was killed – player & bullet interaction and subsequent animation (explosion), gray indicates the end of the local player

The following properties of the simulation was tested:

User interactions (in relation to algorithm runtime interruptions) – event handler does not represent a measurable delay.

Entity-entity collisions (soldier-soldier, soldier-shot) – complexity is determined by the complexity of calculation and optimization of data structures, generally not a problem.

Entity-environment collisions (restriction of access to certain areas of the scene) – depending on the choice of the appropriate data structures and the required accuracy. For example, cross-border control of players' playgrounds was performed once every 100 ms (about 6 to 10 animation frames). This without a visible problem from the player's experience, an FTP increase of 1-2.

User interaction latency during the interaction of a local process with a remote one. The response delay is dependent on the frequency of the message forwarding, at a 30 ms status update, a response of up to 30 seconds for sending the static data, 5 ms for network transmission, max. 30 for sending the response back (sending the status data), 5 ms for network transmission. It follows that the response required after a remote logic process can take up to 70 ms (about 3 animations).

Number of synchronization messages and message latencies.

Scene rendering delays (FPS – max, min, avg) in relation to the number of animated objects – see table 1

Look-ahead for the first level of time synchronization is not relevant.

Differences between local simulation times and global average – the difference is on average several ms, the difference amount decreases with the increasing number of LPs because the increasing sync level 2 algorithm better compensates for the differences.

All the stated properties was tested on simulation model configurations of 8, 12, 20 and 40 logical processes. Measurements took place on identically configured PCs (Intel® Core™ i3-3240 CPU @ 3.40 GHz, 4 GB RAM, Windows 10 64bit, only one application running – Google Chrome browser, version 58). The best performance (FPS plus user experience) is when configuring 20 LPs.

Table 1: Results of tested use case, critical data for simulation run with focusing on animation output and interactive approach

Tested properties	#1	#2	#3	#4	#5	#6	#7	#8
LP count	8	8	12	12	20	20	40	40
User interaction & collision calc enabled	NO	YES	NO	YES	NO	YES	NO	YES
Animation FPS (AVG)	85	80	65	60	33	32	15	11
Sync request count	186	142	137	44	108	119	219	246
Animation 1 frame draw time (AVG) [ms]	34	32	42	42	61	60	92	95
Animation 1 frame draw time (MAX) [ms]	37	50	78	71	79	82	155	180
Animation activity count in animation scene	215	245	278	266	552	548	980	1005

Result notes (table 1):

- User interaction (mouse click, press key SPACE) was programmatically generated. Calculated by uniform distribution (min 400 ms, max 2000 ms) between simulated interactions.
- One animation activity is consists from 5 graphics base elements / primitives (real canvas draw elements) in average.
- Results were collected after 5 minutes (real time) run.

From the table 1, read that the limiting invoice is not the number of LPs (within the specified numbers), but (a) the complexity of the; calculations and (b) the scope (especially in terms of quantity) of the animation.

ISBN 978-88-97999-85-0; Affenzeller, Bruzzone, Jiménez, Longo and Piera Eds.

8. CONCLUSION

The primary motivation for use of web-based simulation is the availability of the runtime environment – web browser – on any computer or modern device connected to a computer network. JavaScript is very well supported by modern-day browsers, and is extensible and well known. This comfort of availability and simplicity is not without a cost – when compared to native applications, the scripts are slow. Web browser simulations can not be compared with native application (using standards like HLA, DIS, TENA, etc. or in general) due to the inequality between compiled languages, multi-threaded access, and graphical output – ie (relatively) direct access to the graphics card.

Altogether, the introduced solution is suited to solving small tasks that do not require complicated calculations and complicated graphical output, but require distributed space or operator workstation. A good example may be the training software operators of the (technological) process, where the distributed approach (different workplaces) is used and complex graphics output is not required – it is just an interactive diagram of the relevant technological process.

Regarding presented use case is at the edge of the technology possibilities. The only possibility of improvement seems to rewrite (source code) the graphics output to WebGL, which allows use graphics card to generate the output. There is the opportunity to reduce the load of current software approach to drawing the output, and use the new available (processor) time for more detailed calculations or larger scale (more entities, events, etc.) simulation itself.

REFERENCES

Fujimoto, Richard M. Parallel and distribution simulation systems. New York: Wiley, 2000. Print.

Kartak, Stepan, and Antonin Kavicka. "WebRTC Technology as a Solution for a Web-Based Distributed Simulation". Proceedings of the European Modeling and Simulation Symposium 2014. Genova: Università di Genova, 2014, s. 343-349. ISBN 978-88-97999-38-6.

Kartak, Stepan. "Web Simulation as a Platform for Training Software Application". Proceedings of the European Modeling and Simulation Symposium 2015. Genova: Università di Genova, 2015, s. 70-78. ISBN 978-88-97999-57-7.

Kartak, Stepan. "Web Simulation as a Platform for Training Software Application". Proceedings of the European Modeling and Simulation Symposium 2016. Genova: Università di Genova, 2016, s. 78-86. ISBN 978-88-97999-76-8.

Voracek, Jan. "Web Simulation as a Platform for Training Software Application". Proceedings of the European Modeling and Simulation Symposium 2016. Genova: Università di Genova, 2016, s. 73-77. ISBN 978-88-97999-76-8.

Kuhl, Frederick, Judith Dahmann, and Richard Weatherly. Creating computer simulation system: an introduction to the high level architecture. Upper Saddle River, NJ: Prentice Hall PTR, 2000. Print.

Tropper, Carl. Parallel and distributed discrete event simulation. New York: Nova Science, 2002. Print.

Kuhl, Frederick, Dahmann, Judith, Weatherly, Richard, Creating Computer Simulation Systems: An Introduction to the High Level Architecture, c2000, Upper Saddle River, NJ; Prentice Hall PTR. ISBN 01-302-2511-8.

Hridel, Jan, and Stepan Kartak. "Web-based simulation in teaching". The European Simulation and Modelling Conference 2013. EUROSIS-ETI, 2013. Print.

The Institute Of Electrical And Electronics Engineers, Inc, 2012, 1278.1-2012: IEEE Standard for Distributed Interactive Simulation - Application Protocols. New York; IEEE. 2012. ISBN 978-0-7381-7310-8.

The Institute Of Electrical And Electronics Engineers, Inc, 1996, 1278.2-1995: IEEE Standard for Distributed Interactive Simulation - Communication Services and Profiles. New York; IEEE. 2002. ISBN 0-7381-0994-0.